UNIVERSITY OF OSLO
Department of informatics

Generating gameplay scenarios through faction conflict modeling.

Master thesis

Marius Brendmoe

1. August 2012

Abstract

A gameplay scenario can be defined as a series of events that emerge from a given context. These events can potentially be influenced by the player through gameplay, which results in meaningful interaction with the simulation. Creating gameplay scenarios in computer games and training simulators is an immensely expensive and time consuming undertaking. A common trait for most scenarios created is that they tend to be static. Once a player has completed the scenario once, he knows exactly how it will behave the next time, reducing or removing the replay value of the gameplay scenario. This thesis investigates how artificial intelligence techniques can be used to define virtual worlds and interaction between entities, such as virtual humans, to dynamically generate gameplay scenarios by simulating the conflict between entities as they clash over conflicting interests in the world.

The first part of this thesis introduces the vast field of artificial intelligence, how it is usually applied in games, and how new concepts are slowly trickling into the field of game artificial intelligence. Topics introduced include crowd simulation techniques, agent simulation and how one can describe arbitrary virtual worlds through the use of semantics, smart objects and fuzzy logic.

The second part describes the practicalities of the implementation. Here, the game engine used to develop the prototype game world is presented and compared to other alternatives. Next, the design and implementation details of the proof of concept implementation, called the "*Faction Interaction Framework",* are described in detail. The design allows for quickly defining the important resources, actions, and potential interactions between entities in a virtual world. Finally, the implementation can be run as an add-on to a virtual world, which can be used to drive scenario generation through conflict simulation.

The work presented in this thesis provides a proof of concept solution for dynamically generating gameplay scenarios. By providing game developers with a pattern for defining the elements of their virtual world that is the source of conflict, the "*Faction interaction framework*" provides an approach to have the virtual world autonomously generate myriads of gameplay scenarios depending on user input. This has potential application especially to large, open world games, massively multiplayer online games and training simulators, where the generation of novel gameplay scenarios is challenging due to the large amount required.

Table of contents

Chapter 1 Introduction

This chapter provides an overview of all the work presented in this thesis. Section 1.1 argues for the importance of the field of study presented here, and provides an introduction to the field of game and simulation research. Section 1.2 contains the problem statement, defining the main questions that this thesis attempts to answer. Section 1.3 describes the limitations applied to the problem for the sake of maintaining a surmountable scope. Next, section 1.4 introduces the research methodology used throughout this thesis, which is followed by contributions through this work, which is presented in section 1.5. Section 1.6 then presents an example scenario which is used throughout the thesis as a point of reference for the examples used throughout the theoretical discussion. Finally section 1.7 provides an outline of the content of each chapter.

1.1 Background and Motivation

Since the arcade games from the 70's and 80's, the face of the game industry has changed drastically. At the time, games were built on dedicated hardware that used jagged lines and boxes to create the virtual world for players to immerse themselves in. Since then, the income and number of people employed in the games industry has skyrocketed [2]. The same is true for the cost of developing a single game.

Since the arcades of the 80s, there has been astounding advances in computational hardware, allowing computer scientists to create breathtaking worlds that sometimes look so realistic that they are mistaken for photographs. With this added realism comes an immense increase in cost. Since Pack-Man was released for the first time in 1980, the cost of developing games has gone through the roof [3]. According to BBC World News, Namco produced Pacman for 100,000 USD in 1982. Halo 3, which was released 2007 cost 15,000,000 USD to create [4]. In 2009, Double Fine released their console title "Brütal Legend" which had a reported budget of 24,000,000 USD [5].

As realism in graphics increase, so do the expectations of realism in interaction. Physics engine technology, artificial intelligence (AI) capable of navigating the complex worlds and reacting appropriately to situations, sound effects and music created by expert composers and graphical content created by a cohort of artists all add up to a very expensive end product. One could go on for a long time about all the features required for a high-end title today, but

we need only to look at the size of today's studios to get an impression of the immense amount of work required.

Because of the daunting number of man-hours invested in these productions, the need for cutting development costs is more apparent now than ever. This need has spawned many companies focused on bringing powerful third-party solutions to the hands of game development studios. These new tools allow developers to focus on creating games instead of spending most of the time creating the underlying technology.

In the last few years, there has been an increased focus on AI in games. This is most apparent in the new middleware solutions being developed and licensed for use in the most prestigious game projects. The Euphoria system [6] by Natural Motion has become widely used in games such as "*Grand Theft Auto 4*" and "*Star Wars: The Force Unleashed*". Havok [7], one of the most widely used physics libraries, has also expanded functionality with an advanced AI module to stay competitive.

While digital games manufacturers are the most prominent users of new middleware, medical and military industries are also investing heavily in simulation research and technology [8]. These technologies are often similar or even the same as the ones used in modern games. The United States MOVES institute is a postgraduate institute dedicated to such research, and in July 2002, they launched the Americas Army computer game for recruitment and training. Kongsberg Defense was in 2010 awarded a contract worth 100 million SEK to develop a ground-air combat simulator for the Swedish Army [9]. In 2008, Pope et al. published a paper describing the challenges and needs for creating realistic agent populations on a large scale [10]. The paper brought attention to the need for more believable background populations in simulations used to prepare soldiers for combat zones. The current conflict in the Middle East requires a whole new skillset as soldiers must be able to gauge the level of animosity in a population and make use of their interpersonal skills to disarm potentially dangerous situations before they escalate into armed conflicts. The generation of such training scenarios are very expensive and are usually scripted to the smallest detail [11, 12]. While this attention to detail is surely necessary to create a realistic simulation, it stands to reason that one static training mission loses its impact once the dialogue options have been explored by the user. Drawing from Pope et al.'s paper on realistic background populations, it would be reasonable to argue that the scenario would become much more flexible if the bystanders could change the outcome of the scenario through the current friction between factions and their current animosity towards the user.

The huge interest for virtual world applications has increased the demand for such simulations to respond in novel and realistic ways. In the real world, the environment and other creatures respond to actions performed by other entities in the system. For such realism to be possible in a game or simulation setting today, designers usually have to handcraft each action and reaction that agents perform. In the games industry, there are examples of this being solved in more novel ways. One approach is demonstrated in the Left 4 Dead series by Valve corp [13]. This series is most noteworthy for their innovative piece of technology called "The Director" which directs the focus of all virtual agents in the simulation, to fine tune pacing and challenge to the skill of the players [14]. Another example is No One Lives Forever 2, where all virtual agents act autonomously, interacting with different objects in the game, creating an illusion that agents were operating with purpose. Russell et al. describe an agent in AI to be *"anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators"* [15] In AI, the real world and a virtual world can both be classified as an environment, the only difference is the model-complexity the agent operates in.

As computer games become more and more realistic, the gap between game developers and engineers developing "serious" simulations is narrowing. As their fields begin to overlap more and more, they find themselves required to address the same challenges. As mentioned in the previous paragraph, one of these challenges is found in creating realistic and immersive conflict scenarios. As it stands now, a large amount of work must be done for each possible scenario that developers wish to present to their users.

Generating these scenarios through "brute force" methods such as scripting, each possible scenario and outcome are becoming more and more problematic as cost of development goes up. In addition to cost, development time is also a factor that leaves current methods inadequate. Soldiers training for missions in the Middle East will soon find themselves repeating the same training missions over and over again without much variation, arguably reducing the effectiveness of the exercise. In the games industry, there are games that rely on AI almost exclusively to provide gameplay experiences, such as "Mount & Blade". However there are very few games that have the budget to implement such complex features as well as the more common scripted narrative. The game "Elder Srolls: Skyrim" is an example of such a game [16]. For the next generation of consoles, both Microsoft and Ubisoft have made comments regarding their focus on dynamic AI solutions [17, 18].

Systems for creating dynamic interaction on an agent level have come a long way, as is evident in the Euphoria software suite [6], among others. Complex solutions that combine physics simulations and character AI to create highly believable world interactions and crowd simulations allow for the creation of scenarios that appear highly realistic. According to Pope et al. [10], this is not enough to generate a believable training environment, and thus, it can be argued that a new approach should be taken. By simulating the interactions between larger groups of agents with varying faction and group allegiances, and allowing them to resolve conflicts in ways determined by the designer, it could be possible to have agents generate a myriad of varied scenarios that would make both computer games and combat simulations more challenging and interesting.

1.2 Problem Statement

In this thesis, we address the challenge of automatically generating virtual world scenarios through faction-interaction modeling. This includes seeking solutions to questions like: *How can one generate gameplay scenarios by simulating the interactions between the agents that populate a game world, while still allowing for scripted events to drive narrative?*

Scripted scenarios in games tend to offer a higher quality of linear narrative and gameplay than can be generated through procedural systems [19]. A scenario in a virtual world can be described as the context in which the game takes place, the development of the game over time, and some events that take place during game play [20]. Creating scenarios is a laborious process, and the creation of such content in large quantities is immensely expensive. Despite the cost of scripted content, it offers little to no replay value. While this might be fine for entertainment products with a one-time fee, it is less beneficial for subscription based games and training simulations where new and different challenges are essential [10]. An additional issue is also posed in that such systems are usually larger in scope than one-time fee games which require considerations to be made in regards to scalability.

This thesis explores ways of assisting game and simulator designers in generating emergent gameplay scenarios [21], while still making it possible to have scripted gameplay sections. Further, by exploring the field of AI, we seek answers for the following: *"What approaches exist? How can these approaches be applied to solve the stated problem? What steps must be made to apply these principles to solutions of massive scale?"*

The goal of this thesis is to further describe the design and development of a software solution that attempts to solve the problem stated above. This framework, called the "*Faction Interaction Framework" (*FIF), will attempt to use modern game development and AI techniques that allow game designers to describe their world and agents to the framework, which in turn, will drive the agents into conflicts which results in novel gameplay scenarios.

To accomplish this, the FIF is based on two concepts within AI research: The first concept is the study and modeling of groups and conflict, commonly described as "Crowd Simulation". While several key concepts will be discussed and drawn upon to create a simulation of group conflict, the most essential piece will be the work of Medler et al. [22] which suggests using models from the field of conflict theory. The second concept which will be at the center of this thesis is the concept of semantic modeling to create a more detailed world for AI-driven systems to use for reasoning.

The FIF will be evaluated by investigating its potential for describing and simulating virtual worlds. This will be accomplished by observing several factors: Does the system actually generate conflict scenarios dynamically? How complex is the framework to use? Given the real-time requirements of training and game worlds, does the framework have a practical application? The scope of these questions is daunting. Due to this, it is essential to focus on viability of the implementation. Therefore, some topics, discussed in the theoretical foundation of this thesis, are excluded from the implementation.

1.3 Limitations

The topics essential for this thesis are focused most intently on creating the scaffolding required to demonstrate the viability of the approach presented. Due to limitations in scope, this causes the exclusion of several extensive topics. The most interesting of these are:

1. *Advanced actor hierarchies*: Actor hierarchies describe the interdependency of factions and their ability to manipulate and order sub-factions to perform different tasks. While the simulation of conflict within factions is a great source of conflict scenarios, this part of the simulation is removed from the scope due to its complexity.
2. *Massive world simulations*: The main reason for this thesis is to investigate a potential approach to generating original gameplay content through simulating conflict. There are few types of software more in need of this than modern massively multiplayer

online games. While this is a very important topic, it is also put outside the scope of this thesis. This thesis focuses on creating a proof of concept for the basic ideas to the approach, and therefore, the work on scalability and creating viable test scenarios would be too large.

3. *Learning AI:* The core reasoning system of the approach proposed in this thesis relies on what can be described as a neural network (if one uses a broad definition of the term). As neural networks can be trained using training sets [15], it would make sense to apply the same techniques to create worlds that followed a specific evolution pattern through training of the actors in the world. This again, is far outside the scope of demonstrating viability, and is therefore excluded completely. However, this topic is discussed briefly in section 8.3 on future work.

1.4 Research Method

The research approach for this thesis is rooted in the methodology specified by the ACM Task Force [23]. This methodology advocates three different paradigms of the computing discipline. Each paradigm is an iterative approach that includes four steps. Each paradigm is applied to a different part of the work, and is iterated as required.

1.4.1 Theory

The theory paradigm consists of four steps followed in the development of valid theory:

1. Characterize objects of study (definition).
2. Hypothesize possible relationships among them (theorem).
3. Determine whether the relationships are true (proof).
4. Interpret the results.

1.4.2 Abstraction

The abstraction paradigm is applied to problems where the model does not agree with experimental evidence. The steps are rooted in the experimental scientific method:

1. Form a hypothesis.
2. Construct the model and make a prediction.
3. Design an experiment and collect data.

4. Analyze the results.

1.4.3 Design

The design paradigm deals with engineering of a solution, and follows the following steps:

1. State requirements.
2. State specifications.
3. Design and implement the system.
4. Test the system.

1.4.4 Summary

The *theory* and *abstraction* paradigms are used mostly when dealing with strict computer science problems with roots in applied mathematics and natural science [23]. This thesis is heavily engineering focused, and thus, the *design* paradigm is used as the foundation for the work presented. Even so, the *theory* and *abstraction* paradigms are essential when evaluating a solution. *Abstraction* is used to analyze and improve upon performance and *theory* is essential for formulating algorithms for the different elements in the specification proposed in the second step of the *design* paradigm.

1.5 Contributions

This thesis has provided an introduction to, and explored the complexities of creating virtual world simulations for games, military and emergency applications. By describing a set of the most used AI techniques for such applications, the reader has been given insight into how different game and training scenarios are built today, as well as the cutting edge solutions that potentially changes this paradigm. Further, these techniques have been used to create a proposal for a framework for simulating conflicts between faction groups. Further, this thesis describes how the proposed framework can be used to describe virtual worlds and assist game developers in creating dynamically changing worlds.

In addition to the theoretical details of game AI and crowd simulation techniques, this thesis describes several game engines and comment on their suitability for use in AI research. Next, a proof of concept implementation of the faction interaction framework is shown in the engine deemed most viable. This implementation is discussed in terms of viability, resource

requirements, scalability and ease of use, should it be used in game or simulation projects. Finally, this thesis discusses the viability of generic AI solutions in regards to game development and discusses potential future improvements for the FIF.

By providing game developers with a pattern for defining the elements of their virtual world that is the source of conflict, the FIF provides an approach to have the virtual world autonomously generate myriads of gameplay scenarios depending on user input. This has potential application especially to large, open world games, massively multiplayer online games and training simulators, where the generation of novel gameplay scenarios is challenging due to the large amount required.

1.6 Example Scenario: Hunger Conflict

In this virtual world scenario, two tribal villages are being given humanitarian aid in the form of food drops from an air-drop program. Every week, a large batch of food is dropped in a field situated halfway between the villages. While the humanitarian organization responsible for delivering the aid has calculated that the food provided should be sufficient to last until the next drop is delivered, tensions have grown as the two villages hoard the food in fear of the food program being discontinued without notice. Each village is led by an elder, who is calm and diplomatic individual, trying their best to mend the strained relations between the two factions.

This scenario is, arguably, a good example of where to apply the FIF framework, as it could potentially be of note in all the different types of virtual world simulations described in section 1.2 (games, military, emergency). One could easily introduce firearms to this scenario, to angle it towards military training, or in the case of games, have two different fantasy races inhabit the villages.

In each village there are various fractions that pursue their own agendas as well as doing their best to aid their village in getting the upper hand. These sub-groups are:

- Thieves: Will always run away from a conflict initiated by other groups.
- Warriors: Has the battle ready trait, causing them to train for violent conflict, and more easily make use of this means to resolve conflicts between factions.

- Farmers: In the extended example, these are capable of producing their own food, however they will prefer to access the more easily accessible food source if it is uncontested.
- Villagers: No specific traits.

In addition, there are villagers with no extended agenda, who's only concern is survival. The warriors of each tribe are advocating the use of violence to gain control of the area where the food supplies are dropped, while the thieves care little either way, and are only concerned with getting as much of the food as they can, no matter what the rest of their village should decide to do.

Each village has its own training area where the warriors of the tribe train to prepare themselves for the conflict they think is coming. The villagers are not worldly folk, and greatly prefer staying within the village boundaries. They especially do not enjoy being in the opposing village.

1.7 Outline

This thesis investigates methods for enriching the knowledge of agents in virtual worlds, generating conflict scenarios and explaining how these contributions can be beneficial to the design and creation of virtual worlds for various purposes. It is divided into several parts, where Chapter 2 and 3 compose the theoretical part which introduces key topics in AI. The next part discusses the technicalities required for implementation of framework and prototype, this information is presented in chapter 4-6. Finally, an in-depth discussion of the results is presented in chapter 7 and 8.

We start by providing an overview of the terminology used when discussing game engines in section 2.1. There are several core elements that come into play when designing AI for games [24]: movement, decision making, strategy and agent reasoning. All these elements are discussed in section 2.2. Further background information is provided in a discussion on the effects of spatial partitioning and its potential effects on real time simulations in Spatial partitioning

Once the most essential terminology has been introduced, we move on to the more detailed introduction to advanced techniques in the field of Game AI. As discussed in section 1.4, we

will investigate methods for describing the world in any detail. World representation techniques and how they apply to the FIF are discussed in sections 3.5 and 3.7. The other core concept of the FIF is that of conflict modeling and crowd simulation. These topics are discussed in, sections 3.3 and 3.7. In addition to this, we introduce concepts essential to the construction of a game scenario, such as agent reasoning systems in sections 3.1 and 3.2 as well as advanced navigation topics described in section 3.9.

Given this solid introduction to AI concepts, the next chapter investigates potential game engines for use in AI research and game prototyping. Section 4.1 provides an overview of several game engines, while section 4.2 presents a more thorough discussion of the Unity3D engine. This investigation of potential game engines is very important for keeping within a reasonable time schedule. The engine will be used in conjunction with the FIF to produce a game scenario as a proof of concept. The design proposal for the FIF is described in Chapter 5. Here, all aspects of the core framework implementation are discussed. The world representation implementation is described in sections 5.1, 5.2 and 5.3. Faction and conflict systems are laid out in section 5.4, 5.5 and 5.6.

Chapter 6 provides further details into the practicalities of implementing the FIF prototype. First a short overview is provided of the language used in section 6.1 before the details of each FIF feature is described in the remaining sections. This chapter leads directly into Chapter 7, which describes the implementation of the test scenario in Unity3D, and how this was integrated with the FIF.

Chapter 8 concludes the thesis. Here, a quick summary is provided in section 8.1. Next, a brief mention of contributions to the field is listed in section 8.2. Finally, a discussion regarding future work is presented in section 8.3. Here, some concepts that were left out of the prototype for the sake of simplicity are discussed in detail, alongside concepts for parallelism, scalability and applicability to large scale multiuser environments.

Chapter 2 Background

This chapter introduces terminology and concepts that form the basis for discussion of real time simulation and game development research. Section 2.1 provides a brief description of how a game engine functions and describes terms commonly used in the field. Understanding of this terminology is essential for understanding the more complex topics covered in later sections. Section 2.2 introduces the fundamental terminology and concepts in the field of AI. The topics presented, are essential for understanding the complexities of the more advanced methods introduced in Chapter 3. Finally, section 2.3 describes spatial partitioning as a general principle for optimizing allocation of computational resources.

2.1 Game engine overview

Game engines are highly complex systems constructed from many different modules to

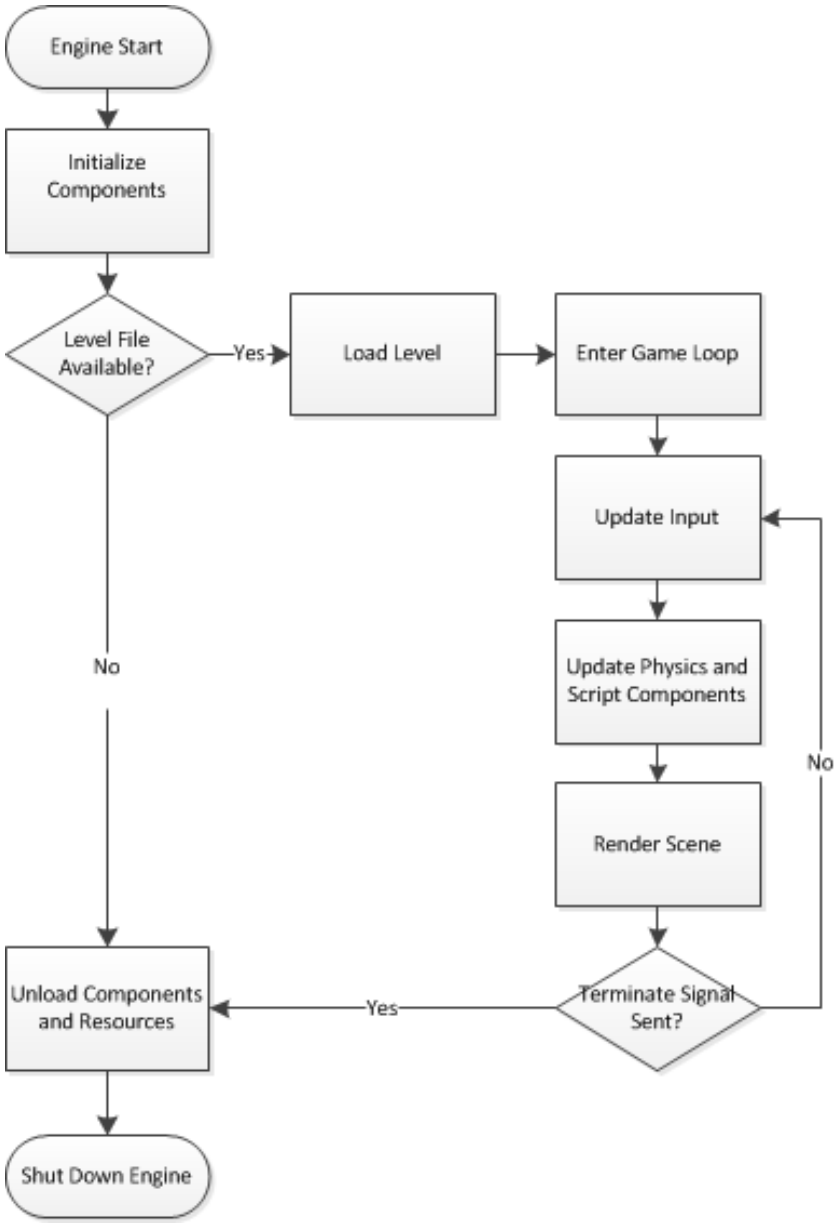


Figure 1 - A simple game engine loop

provide a wide range of services to game designers. Game engines are real time systems that usually run on an update frequency of 30 or 60 frames per second. Sometimes different components run at different update frequencies or in parallel [25]. Figure 1 shows a simplified game engine loop, where a single frame can be seen below the element “enter game loop”. It should be noted that game engines and training simulators more often than not, share the same requirements and solve many problems in the same fashion. Therefore, arguments that are made from a game development perspective can often be directly applied to training simulators. These systems are usually organized into modules, or components [26]. A typical set of such modules is listed in the following subsections.

2.1.1 Scene Management

The scene manager is responsible for organizing the world in which the simulation takes place. Scenes are usually arranged in trees, with the scene itself as the root. Most game engines rely heavily on hierarchies to organize the game play logic [26]. It is common to utilize polymorphism to construct advanced combinations of features that form complete simulated objects. Figure 22 shows what such a hierarchy might look like: here you can see the basic game object and dynamic game object classes containing the essential information

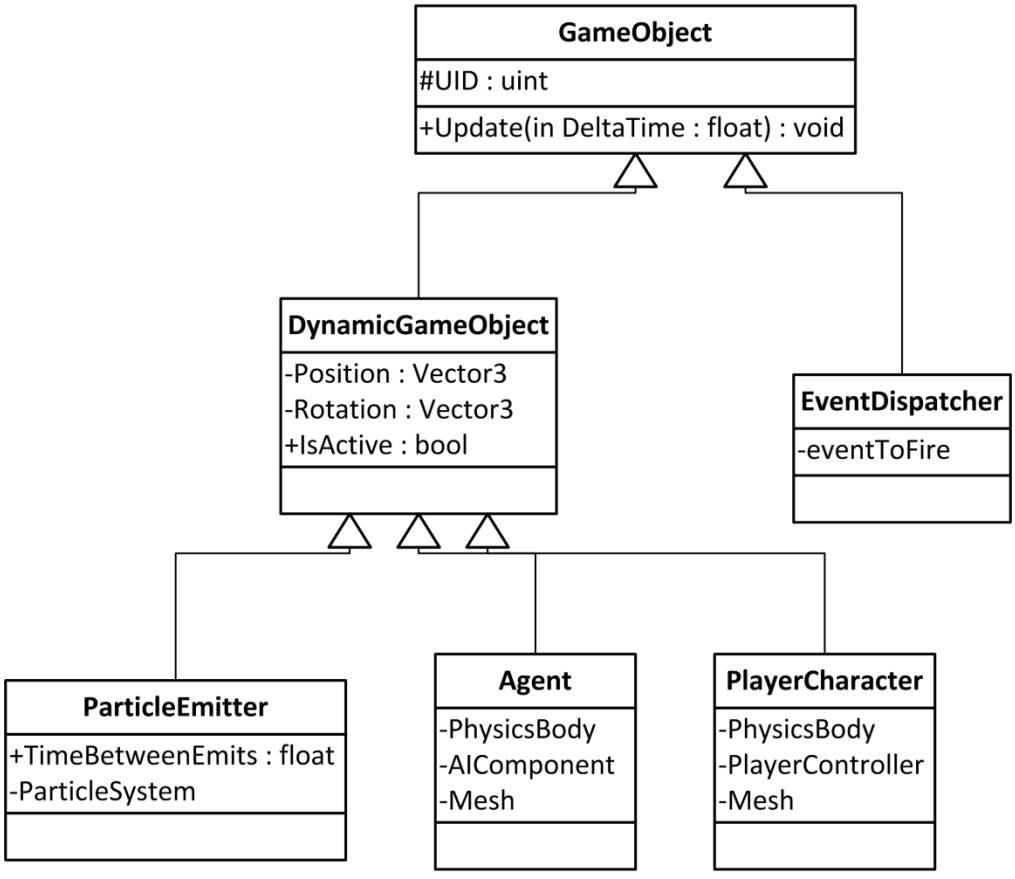


Figure 2 - Simple Object Hierarchy

required to exist in the virtual world. Inheriting these features are then the agent and particle emitter classes. The particle emitter is used in rendering to enhance visual fidelity, while the agent is a simulated entity in the game. The agent contains a physics body which allows it to be affected by the physics system, an AI component that evaluates the world and attempts to manipulate it, and finally a mesh which is used for rendering the object. The third entity inheriting from the dynamic game object class is the player character. This class contains most of the same logic as the agent, but instead of an AI component that controls the other systems, the player character has a controller that takes input from keyboard or other physical devices. Objects designed in this fashion are then put in a tree-structure that can be traversed to relay messages to child objects, gain information about objects' relation to each other, relative position, etc. An example of how this works is shown in Figure 33, which describes the process of creating a 3D model of a man in Carnegie-Mellon's Panda3D engine [27].

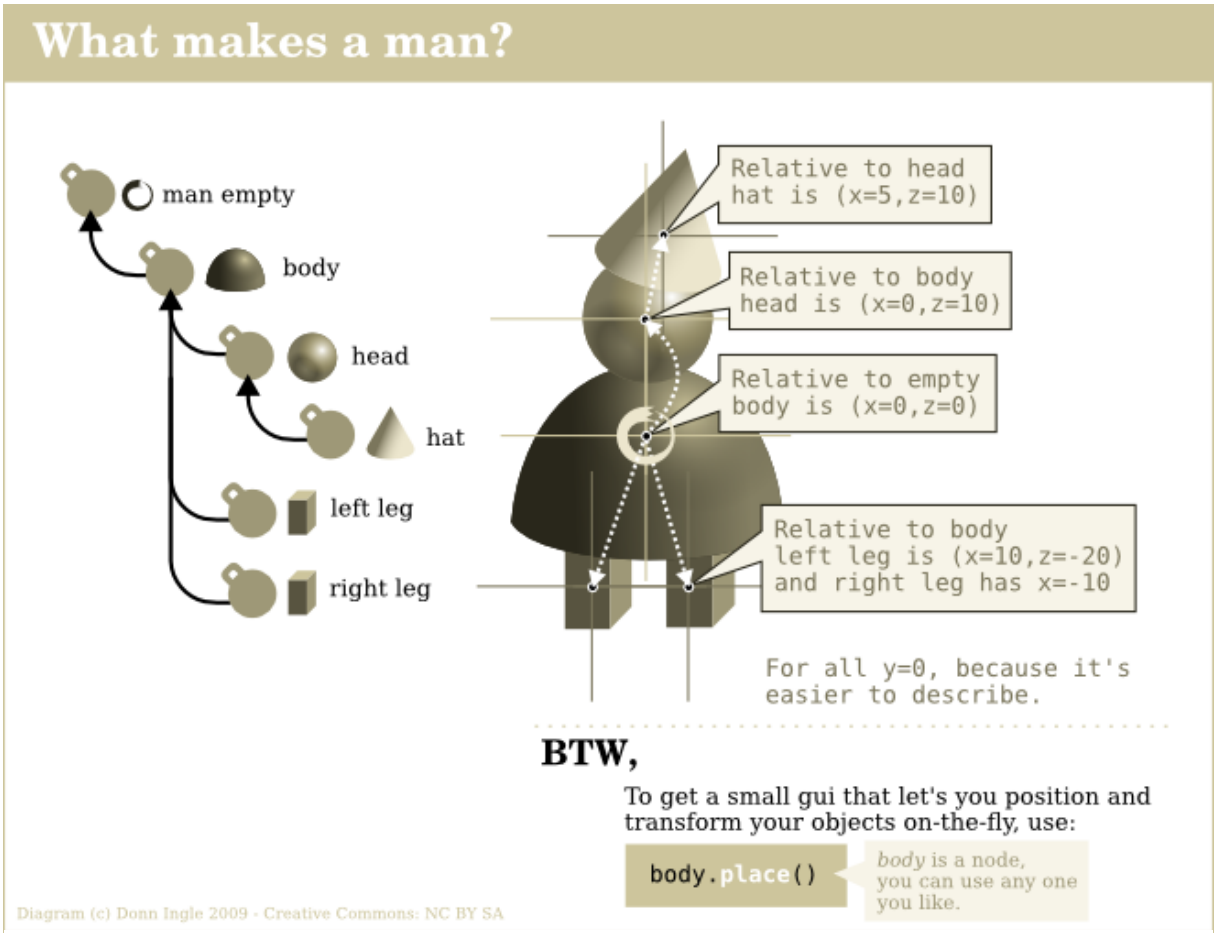


Figure 3 - Scene Graph Example [1]

2.1.2 Logging and Error handling

As game engines are highly complex software systems, it is imperative to have access to detailed information about runtime interaction between entities. A logging component usually contains some or all of the following features [25, 28]:

- Write message to console
- Write message to file
- Perform dump of stack trace to file on crash
- Performance monitoring

2.1.3 Content Management

Content in games refer to all types of files that impact the game in any way. Large parts of modern game engines are concerned with importing and converting different file types and formats into workable internal data types that can be used by other components [26]. Typical file types handled by a content component are:

- Text files
- Fonts
- Sound/Music files
- 3D models
- Texture files
- Game scripts

2.1.4 Physics

Physics simulation is an integral part of many modern games. This component usually performs all work that deals with collisions and applies physics to all game objects defined in the scene manager. Physics is commonly a third party middleware solution shipped with the engine such as Havok or PhysX. Because of this, it is common for physics components to be designed as a black box. Physics components often require higher update frequencies than other components, running at 60 to 120 frames per second.

2.1.5 Rendering

Rendering components deal with combining spatial game information and graphics resources into images on the screen. These components can deal with either 2D or 3D assets. Rendering is not covered in any great detail in this thesis, but is listed here for the sake of completeness.

2.2 Artificial Intelligence

In their book Artificial Intelligence a modern approach, Stuart Russel and Peter Norvig explain that there are two definitions of AI [15]. The first definition states that AI is the study of how to make computers act and think like humans, or as stated in the book *"The art of creating machines that perform functions that require intelligence when performed by people*". The second definition states that AI is the study of how to create machines that behave rationally, again quoted from the book by Russel and Norvig; *"AI ... is concerned with intelligent behavior in artifacts"*. These two approaches can again be split into two concepts: acting and thinking. This section gives a brief introduction to AI and how it applies to games. Advanced AI topics are discussed in Chapter 3.

2.2.1 Acting Humanly

The Turing test was proposed by Alan Turing in his paper "*Computing Machinery and Intelligence***"** from 1950 [15]. The test states that a computer can be said to be intelligent if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or a computer. Natural Language processing, knowledge representation, automated reasoning and machine learning all fall into this category. Building artificial intelligence with this focus is referred to as the "*Turing test approach*" by Russel and Norvig [15].

2.2.2 Thinking Humanly

This approach attempts to replicate the way the human brain works on a more basic level and is closely related to cognitive science. General problem solvers and neutral networks fall into this category.

2.2.3 Thinking Rationally

Referred to by Russel and Norvig as "the laws of thought approach", this field of study attempts to build on logic formalism to create a program that can solve any problem through logical induction.

2.2.4 Acting Rationally

"The rational agent approach" attempts to construct agents capable of modify their world to fulfill some goal in a way that "makes sense". Russel and Norvig argue that this approach has the most direct application as rational agents are capable of completing goals that can be easily defined in a language that computers can understand today (e.g., "Temperature is below 20 centigrade, turn on heating"). While the skills required by rational agents are useful for completing the Turing test, the goal of such agents is not to complete said test.

2.2.5 Game and Simulation AI

AI used in computer games and simulations can be classified as either Agents or Virtual Humans. *Agents* belong in Russel and Norvig's rational agent approach. The goal of agents in games and simulations is to perform a special function that will challenge or assist the user in some way. In the game Gears of War, by Epic Games, the "Covenant" warriors are not supposed to give the impression of humanity, instead they are targets for the player to shoot. In a simulator for missile defense systems, the enemy planes are also agents, as it might be more valuable for soldiers to practice against perfect targets than humanlike behavior.

Virtual Humans attempt to exhibit humanlike behavior. In games, these might be virtual players competing against you, e.g., in the game of Unreal Tournament [29] or an opponent in a strategy game. In simulators for low intensity conflicts and firefight scenarios for ground troops, all opponents and representation of inhabitants need to behave as close to human as possible to give a realistic training scenario [10].

In addition to the challenge already posed by attempting to resemble human behavior, game and simulation AI also has tight time constraints to have any chance of delivering a believable performance to the spectator. A virtual human that takes a minute to react to gunfire would do little to increase the realism of the scenario. Because of this constraint, games and simulations tend to lean heavily towards the *rational agent approach*, and it is this that will be the main focus of this thesis. However, ideas from the *Turing based approach* will also be considered.

2.2.6 *Sensors*

Sensors can be anything from cameras, light sensors and antennas in robotics, to ray cast functions and path finding queries in a game engine. The data gathered by these sensors are first analyzed in the sensing stage, before any useful information is used to inform the world model. It should be noted that the sensing stage can run asynchronously from the rest of the simulation, and different sensors can gather data at varied rates. For instance, an agent could have sensors constantly probing the virtual world for objects that it needs to avoid for collision avoidance purposes, while it might seldom query the sound engine for any sound events that could be of interest.

2.2.7 *The world model*

The world model is the basis of all decisions performed by the agent. This model varies greatly in complexity even inside the same simulation. Using "*Hunger Game"* one could imagine that a normal villager has a very simple understanding of the world and how to traverse it, while a thief has a more complex model of shadows and trenches to hide in. A warrior might have additional information in its world model, describing where to find weapons, which agents to fight and avoid, patrol routes around his village and information about dangerous areas where his alert state should be increased. This world model is used by some reasoning engine to decide what actions will be taken next.

2.2.8 Actuators

Actuators are the means by which the reasoning engine affects the world around it in the hopes of changing its world model to one that the agent finds more agreeable. This can be a messaging system for which to communicate with the world, the movement part of a path graph or manifestations of limbs that allow the agent to manipulate objects.

2.2.9 Reflex-agents

Reflex-agents are the basis of many modern virtual worlds, robots and simulations. Reflex-agents are composed of *sensors* that allow them to perceive the world around them, *actuators* that enable the agent to interact and modify the world, and finally, the internal *world model* which describes the world in the "mind" of the agent [15]. Figure 4 describes how sensors, world model and actuators together form a reflex agent.

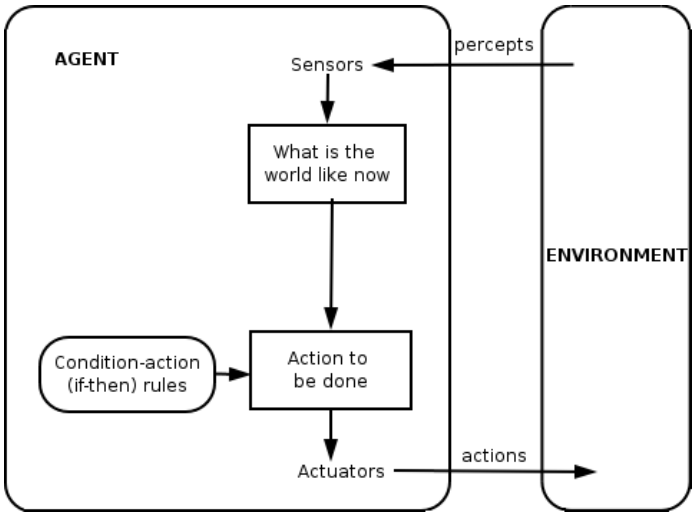


Figure 4 - A simple reflex agent

The environment in Figure 4 represents the totality of all values in the virtual world, or in the case of robotics, the real world. Percepts are collected by sensors, which updates the world model, represented in the figure as the process "What is the world like now". Some sort of reasoning engine decides upon what action should be done, which results in actuators being activated to perform said action. This will (hopefully) change the state of the world, which will again be fed back as percepts into the sensors, completing the cycle.

2.2.10Summary

AI is a very broad definition for a large set of techniques that can be applied to solve a myriad of problems. When building AI for games and simulations, it is fairly common to rely on a small subset of all the tools in the AI toolbox. While acting humanly certainly comes into play for virtual humans and agents in games and simulations, they usually have so little screen time that this AI challenge is reduced greatly compared to a real Turing test. The main focus is therefore often on building agents that can act rationally as is the case with the *simple reflex agent*. In this thesis, there will also be some focus on the concept of thinking humanly, as arguments will be made for how this can be utilized to create interesting forms of gameplay and simulations.

2.3 Spatial partitioning

A spatial data structure is one that organizes geometry in some n-dimensional space. These structures are very useful in optimizing queries on datasets in games and simulations as most of the objects in a game world are spatially persistent or semi-persistent. As most game and simulated objects that persist over longer periods of time also tend to move at a relatively slow velocity, even these objects require fairly little computation time to maintain. The data structures used for spatial partitioning are most commonly hierarchical, because of the reasons stated above. This means in turn that localized queries get significantly faster, typically improving from O(n) to O(log n). [30]

The most common types of spatial data structures are bounding volume hierarchies (BVHs), quadtrees and octrees. BVHs work by putting complicated objects, such as a triangle mesh, inside a simplified representation such as a bounding sphere. This bounding sphere is then used as base for the initial query, should the query fail (as it does on most objects) the more complex data does not have to be considered. Quadtrees and octrees are very similar data structures that create a tree representation of the virtual world by creating recursively smaller axis aligned bounding boxes (or rectangles in the case of quadtrees). These bounding boxes are created by taking the root box and splitting at the center into additional four (or eight) new boxes. Once created, it is populated by all the game objects in their respective child nodes and as the simulation progresses they are moved from one node to another. Localized queries to such a solution are highly effective as one can query only the leaf nodes of adjacent boxes.

While it is common to populate these data structures by only having objects occupy leaf nodes, and duplicating objects that span more than one leaf, there are other approaches possible as well. To avoid duplication of data it is possible to move objects that span several leaves up the tree to the first branch capable of holding the object in its entirety. This has been known to cause performance hits when dynamic objects are positioned at the center of the octree, propagating them to the root node. This problem can be trivialized by carefully considering the layout of the static objects in the scene [30].

Spatial partitioning will most likely be required to control the simulation scaling. As the groups of agents in conflict grow larger, the resource constraints of the system will begin to show. To conform to the real time requirements stated earlier, it will then be required to scale back the level of detail (LOD) of the simulation in areas not under scrutiny by the users. Spatial partitioning strategies are reliable and well established for such use.

2.4 Summary

This chapter has introduced the most fundamental concepts required to follow the remainder of this thesis. A basic grasp of how a game engine works "under the hood" was introduced in section 2.1. The concepts and terms discussed there will often be mentioned when discussing implementation details and challenges. This chapter has also provided quick look at the most basic concepts within the field of AI. These concepts are fundamental to the understanding of Chapter 3, which introduces more complex AI paradigms that require the basics discussed here. Finally, a brief introduction to spatial partitioning and its impact on computational resource management has been provided, as a basis for further discussion on scalability in future chapters.

Chapter 3 will make use of the concepts discussed in this chapter to introduce more complex AI methodologies for crowd simulation, world representation and reasoning. Combined, these two chapters provide the foundation for the design and practical implementation discussed in chapter 4-6.

Chapter 3 Topics in artificial intelligence

This chapter describes different topics in artificial intelligence and how these topics apply to games. The field of AI is vast and theories are borrowed heavily from psychology and social sciences. This chapter will explain the various topics that must be considered when attempting to model interaction between factions and single entities, which is used to inform the design of the FIF.

Section 3.1 and 3.2 introduce two common methods of structuring agent reasoning systems to handle different world scenarios. These principles will be applied to build agents for the proof of concept scenario. Section 3.3 introduces concepts for crowd simulation, which in turn is extended for simulating distributed entities in the form of factions and groups in section 3.4. These sections are used to facilitate the "internals" of agents in the system, or in other words; the systems responsible for simulating needs, desires and prioritization of actions.

While the first part of this chapter deals with simulating the internal processes of an agent, the next step is to introduce techniques for creating a detailed virtual world that can have reasoning applied to it. The way world information is structured is described in sections 3.5, 3.6 and 3.7. Next, some attention is given to introducing neural networks and how they inspired the core reasoning system of the FIF in section 3.8. We then discuss navigation in section 3.9 as this is always a requirement of real time simulations involving autonomous agents. Finally, we present some techniques for scalability in AI and game design in section 3.10, which adds to the completeness of the discussion in regards to modeling large factions.

3.1 Agent reasoning: finite state machines

The finite state machine approach to agent reasoning is one of the easiest to implement on a smaller scale. A finite state machine (FSM) is a system of states representing the internal direction of an agent at a given time. At any given time, one state is active, and the code contained within is executed. Each state is capable of directing all actions for the agent for a period of time, organizing tasks such as animation, path finding and target acquisition. During each update of the state, it will run through its list of possible state transitions. Should a state transition be triggered, the current state will suspend itself and set the triggered state as current. A state can also be triggered from outside the state machine, by a scripted event or similar.

It is common to arrange several smaller state machines into a hierarchical state machine. This ordering is used to avoid state duplication. Going back to the "*hunger conflict"* example, a villager from the southern village might be heading home with food, when confronted by an angry villager from the northern village. When confronted by this individual, the southern villager would have to postpone the current state of bringing food to respond properly to the confrontation. For a "normal" state machine, there would be no way for the employee to resume its old state of bringing food to the village. With a hierarchical state machine, the "argue" state would be attached as a child state of the bring food state, and would allow the state machine to propagate back up the tree to the "bring food" state once the argument was resolved [24].

Hierarchical state machines have become one of the most widely used approaches to creating complex behavior in computer games [31]. The ability to transition into and away from hierarchies of state machines depending on the current game context is a powerful tool for designers to tailor game behavior. This is also a highly useful property in regards to modeling conflict systems, as it would allow for a varied set of conflict resolution strategies to be combined into a larger state machine hierarchy. This way, the state machine could be created through a combination of framework modules and state machine specifically tailored for certain scenarios.

A known problem with state machine approaches is that managing the state transitions becomes increasingly difficult as the number of states and state machines increase [31]. This could prove to be a challenging obstacle to overcome in a framework that is to support scalability. Because of this issue, it might be more feasible to implement agent level controllers using behavior trees. However, for showing how the thesis framework can be applied to gameplay, the limited example system might not be subject to these complexities and thus, FSMs might be a reasonable solution to the problem.

3.2 Agent reasoning: behavior trees

Behavior trees have become a highly popular approach for game developers to create reasoning systems that cope well with the changing needs of an organically growing system. Behavior trees attempt to provide a synthesis of the best features from other approaches such as state machines, schedulers and planners [24].

On the surface, a behavior tree looks very much like a hierarchical finite state machine. While states are self-contained action patterns that can stand by themselves, behavior trees attempt to use a finer granularity to specify building blocks from which to create said patterns. Because of this finer granularity, it is easier to get a minimum of behavior active in the simulation. All building blocks used to construct a behavior tree are called tasks. The most basic tasks in a behavior tree are *conditions*, *actions* and *composites*.

Conditions are tasks that perform some kind of data comparison, such as "is there food left on the pallet" or simply "am I in range of that object". *Actions* are tasks that alter the game world, such as playing an animation or sound. Finally, composites are combinations of the two previous task types. *Composites* can be sequences of tasks, running each one in order, or a selector that picks what child task to run. Actions and conditions both represent leaves in the tree, while composites define branches.

All tasks derive from the same interface, allowing each one to be an isolated feature that can be combined freely through the use of composites. All tasks report failure or success depending on the result of their runtime. Composites will then decide what to do with that information, either to attempt resolving the failure inside its given behavior-space or propagate it up the behavior tree.

In addition to these basic tasks, there are decorators and parallels. *Decorators* are inspired by the "decorator" software design pattern, which suggests wrapping a class around another class to improve or change its functionality. This idea is taken into the behavior tree, by creating a branch node with only one child. The decorator will then modify the behavior of its child node in some way, for example monitoring its use and disabling it when it can no longer be run. The *parallel* is a composite that allows its child tasks to be run at the same time. It keeps track of all the children running and has an internal understanding of how to react when one of the children return while the others are running [32].

This fine granularity approach to building behaviors allows for AI to be built "bottom up", creating simple modular tasks for functionality that is required by all systems, such as path finding, range checks, walk animations and similar. Once these important building blocks are in place, more advanced behaviors can be built as they are required. By defining the structure of the behavior tree in an online language such as LUA [33] or through XML, one could even change the behavior of an agent while the game is running, allowing for simulations to be modified in real time.

By creating a set of baseline functionalities for high level decision making and reasoning, that utilize abstract conflict resolution strategies, behavior trees can be created in such a fashion that they allow content developers to implement all their own behavior on an agent level, while still conforming to the design patterns imposed by the framework in order to generate valid conflict scenarios.

3.3 Crowd simulation

Modeling of crowds and the interactions between individuals therein, are essential topics of the modern world as they allow for studying the behavior of crowds under varying circumstances. The field comprises objectives such as modeling interaction between individuals of different emotional states such as simulating psychosocial behavior in non-player characters [34], to simulating crowd behavior in crisis situations such as escaping a building on fire [35]. These solutions are usually focused on maintaining realism down to the individual level at all times and is thus constrained to small numbers of virtual agents unless the demand for real time simulation is dropped.

Zhou et al. provide an overview of different crowd modeling technologies [36]. The information used to classify the different simulation technologies are crowd sizes and time scale. The authors argue that there are three different approaches to crowd modeling.

3.3.1 Flow based modeling

Flow based modeling focuses on the movement of thousands of people and makes use of fluid simulation techniques to create the wanted effect. This approach provides nothing in terms of advanced behavior for individual ages, and therefore was not considered as a viable option for the problems investigated by this thesis.

3.3.2 Entity based methods

Making use of ideas from particles physics, each individual is treated as a "particle" in the system. Each particle is subject to a series of "social and physical forces" from a combination of local and global generators. As mentioned earlier, simulating groups in emergency situations such as a burning building is a typical use of crowd simulators. Braun et al. tackle this exact problem in [35], where they create repelling fields to represent areas on fire and attach attracting forces to more sophisticated agents that represent leaders capable of directing crowds to safety. In addition, they add force generators to represent influence inflicted by

panicking individuals and repelling fields keeping entities at a certain distance from each other. This method has also been used to replicate emerging phenomena such as jamming and flocking. According to Zhou et al. in [36], this method falls somewhere between flow based and agent based approaches to crowd modeling, both in terms of time span and crowd size.

3.3.3 Agent based simulation

Agent based methods model each individual as a complex system with internal emotions, needs, wants and utilizes advanced forms of interaction with other agents. From these interactions, new relationships and changes in the attitudes of the whole group emerge. Bailey et al. used this approach to model the effect of introducing a negative or hostile individual into a group [34]. Their implementation defines internal states such as emotions and personality, as well as social information for each character, such as ties between individuals, group memberships and social influence modifiers. In addition to these psychological and social constructs, is the notion of current situation or context; to create believable simulations, all agents must have an understanding of their current situation. In his article [37], on agents that can relate to physical spaces, Adam Russell draws attention to the lack of context in digital games today, and how this is of critical importance to the progress of believable agents. A typical problem related to situational awareness is to determine the change in behavior when a physical space transitions from a safe area to a danger zone. Using the example of "*hunger conflict"* as a basis, assume that the conflict has escalated to a state where armed aggression has been initiated by one side of the conflict. The unarmed population present in the field would naturally respond by fleeing the area upon recognizing the armed agents of the opposing village. In a setting where an area has potential for transitioning from one theatre to another, it is extremely important for the agents in the theatre to respond correctly. If agents simulating the general population fail to grasp their situational transition, they would continue with their daily order of business while the tension between fighters escalated to a violent struggle. Zhou et al. propose that agent based approaches should be used for most long term simulations due to the requirements listed, and notes this approach as the most common for digital entertainment products [36].

3.3.4 Summary

When it comes to modeling agents and the social interaction between them, the main roadblock is the amount of computational resources required; especially when the system has real-time requirements, such as digital games or training simulators. Consequently, one must

consider what level of detail each agent can utilize without diminishing the realism of the system while still operating under real-time constraints. It will arguably be beneficial to implement more than one of the aforementioned crowd simulation techniques, in some sort of level of detail (LOD) system, depending on the current requirements of the simulation and load on the system. The implementation proposed in Chapter 5, relies solely on agent based simulation, as the most important features deal with detailed world representation and conflict generation. These features require more advanced simulations from the agents than can be provided using any of the other models. However, it can be argued that if one was to scale the implementation to massive proportions, for instance to simulate entire countries, one could utilize entity based methods to simulate individuals in the population while agents were responsible for the actions of fractions within the nation.

3.4 Factions and groups

While crowd simulators usually are concerned with the behavior of individuals in a spatially localized group, there is also need for a way to model the less concrete bonds between individuals. Groupings of different kinds can have an effect on an individual's beliefs, behaviors and their interpretation of what situation they are in. Going back to the example of the "*hunger conflict"*, the armed agitators would be in a completely different situational context than the unarmed villagers. In addition, some of the unarmed villagers might also have affiliations with the armed agitators and thus choose to join this group instead of responding in the same way as the other unarmed villagers. In other words, some agents may change their situational context depending on the other agents present in the area.

Medler et al. propose a generalized Conflict Theory as a basis framework to model these interactions [22]. In their paper, they define three types of resources: wealth, power and prestige. Conflicts arise when individuals with incompatible goals interact with each other. Examples of such interaction are: deprivation of a resource type, illegitimate power, role incompatibility and belligerent actions.

Role incompatibilities occur between individuals of equal power or when one individual attempts to leverage power over another. An example of equal power conflict, or horizontal conflict, is a group of students arguing over how to design their project. Vertical conflict could occur when a soldier is commanded by his officer to do something morally questionable.

Medler et al. further propose a model for group interaction that consists of three "layers" [22]: Agents are individuals with their own needs and desires, groups are sets of individuals and actors are sets of groups with aligned goals. An actor can consist of one or more groups, and a group can consist of one or more individual agents. Note that an individual can be a member of several groups, and groups can also be part of several actors.

As an example of this organizational idea, consider the "*hunger conflict*", where one could define the entire village as a single actor with each part of society (farmers, thieves, general population) as groups within the actor, and the warriors as its conflict organization.

The framework described in Chapter 5 also includes a world model and a conflict behavior model. The world model describes the relations between actors as well as the needs and desires of these. The conflict behavior model describes all available actions the actor can take to further their goals. In other words, the world model describes how conflict can occur, while the behavior model describes how conflicts can be resolved.

The concepts introduced by Medler et al. [22] provide a good basis for the data and systems required to model an advanced conflict model. By using their definition of resources, actors, groups and agents, the core data model for the framework is established.

3.5 Smart objects

As virtual worlds become increasingly dynamic and the need for post-launch modifications increase, it is essential to find good ways to organize an AI framework so that the content base of the simulation running it can be expanded without the need for a significant rewrite. The basic approach for driving AI decision making and world interaction is to root this firmly within the virtual bounds of the agent. With smart objects, the object itself contains much of the information required to drive action and planning for the exchange between agent and object [38].

The main concept of smart objects is that each object is in itself a supplier of one or more services [39]. For example; a field might register itself as providing the provide food service, as well as "landing site" while a torch would register itself as a provider of light and potentially heat. An agent trying to find a way to become less hungry would query some central system, (or internal memory map, depending on the implementation) where it would be directed to one of the closest sources of this service. One of the best known uses of smart objects is the critically acclaimed game series "The Sims" [40].

"The Sims" [41] is a life simulator, which allows the player to create their own family of "sims". If left unattended, these sims will lead their own life, attempting as best they can to fulfill their desires. The player can then attempt to manipulate the life of their sims, by finding jobs for them, buying new things for their houses and introducing them to other sims in the neighborhood. "The Sims" uses smart objects for all game objects that can fulfill one of the desires of a sim. By using this technique, the game developers (and the highly creative mod community) could easily add new objects that sims could interact with, without having to change the game code.

In addition to broadcasting their services, smart objects can contain a myriad of additional information that allow for very powerful problem resolution approaches [42]. Once the object responsible for supplying some service is contacted by the agent, it can initiate a negotiation cycle to find if it is possible for the object to indeed supply the resource to this specific agent. In a virtual world that allows for extensive refactoring and expansion of content base, there might be several prerequisites that the object could require to make sure that no odd behaviors occur. Smart objects often list a set of valid animations for interacting with the object, as well as where the agent must be located before it may begin playing these animations in the first place. More advanced objects are embedded with whole action plans for their use, which they pass on to agents contacting them about their services [43].

The plans provided by objects could be as simple as hinted to above, listing simply a position for the agent to be in as well as what animation to play. A plan could also be more advanced, giving the agent a set of goals that must be completed before the service can be utilized. In the morbid case of the stray dog as a source of food from the previous section, the dog as a smart object could provide a plan that would require the agent to slay skin and cook it before becoming less hungry. These plans also allows the agent to weight the potential gain against the cost of completing the steps of the plan provided, allowing the reasoning engine to choose a service supplier with an acceptable effort cost.

As a more detailed example, consider the *"hunger conflict"* example once again. One could imagine a field that lists itself as a supplier of "food". The field would respond to an agent's request by initially checking its internal state to see if it is "sown", if not, it would look through available "sowing" objects nearby and add this to the agent's plan. The agent would then attempt to calculate the cost/gain relationship for sowing the field before being provided with food. Additionally, the field would also add "water" and "harvest" to the agent's plan. Each step would have a cost associated with it, which would form a total cost for which the

agent would use to compute the total potential gain. Having computed the complete cost of accessing the service, the agent could potentially investigate other objects offering the "food" service to see if the cost versus gain analysis would yield a better result given their internal state

Smart objects allow for new objects to be added as they are completed in development, without having to make any modifications to the underlying AI engine. The smart object keeps track of what animations the agent must have available to it to interact, as well as what services the agent can utilize by interacting with the object. This way the object simply has to register with the service provider interface used by other objects and the agent will automatically be able to interact with this new addition to the simulation.

The smart object design approach is essential for development of a scalable framework, as it allows for defining interfaces that content creators can use as guidelines for meshing their new features with the framework functionality. Using smart objects to define resource objects as well as conflict resolution areas will allow for quick expansion of the feature set in a given implementation.

3.6 Fuzzy logic

Fuzzy logic extends the traditional principles of logic by allowing for varying degrees of truths to be defined. A predicate in logic is either true or false. For instance, a player could be hurt or not. In fuzzy logic, a predicate is instead defined with a corresponding value representing to what extent the predicate holds true. A predicate set is the entire range of which a predicate can be defined. The value an agent has describing its relation to a predicate is most commonly referred to as *degree of membership* [24]. It is common to use a floating point value ranging from 0.0 to 1.0, including any representable number within this range. This does not mean that these values should be treated as percentages however [24].

Fuzzy logic operates on the premise that all values representing degrees of membership are normalized. Usually, this is not the case for data collected from outside the reasoning engine, and therefore a method known as f*uzzification* is used to bring all variables into the same range. In essence, this is accomplished by membership functions, which bring the input variables into the range used by internal operations. These functions can take any shape required to bring the data into the range required by the reasoning engine. An example can be seen in Example 1. *Defuzzification* is the process of turning the internal data back into a useful

format for whatever system is making use of fuzzy logic inference. Depending on the expected outcomes from using fuzzy logic, there are a set of different solutions to choose from. The simplest form of *defuzzification* is simply to take the *highest membership* relevant to the output function.

An agent's movement speed is affected by the game values [encumbrance, health, speed]. The fuzzy logic engine used by the game has membership functions designed to translate these values into values representing degrees of membership to each part of the predicate set [crawl , walk , run]. The membership functions are arbitrary, however they must yield a result corresponding to the range defined by the fuzzy logic engine. Assume that these functions yield the following result for an arbitrary agent [crawl = 0.2, walk = 0.4, run = 0.7]. Highest membership defuzzification concludes that the agent should be moving at running speed.

Example 1 – Example for highest membership in fuzzy logic

The downside of using highest membership as is shown in Example 1 is that this method only considers the highest value and thusly fails to consider the effects that might be incurred from the other predicate sets.

A common approach in computer game AI is to use a *blending based on membership* approach. This is simply to calculate the sum of the normalized degree of membership values multiplied each by the corresponding max output value. Using the crawl, walk, run example in Example 1, we would have the corresponding max output values *[crawl = 0.5 km/h, walk = 3. Km/h run = 6 km/h]*. Firstly, one would normalize the degrees of membership from before, then sum the new values for the degrees of membership and multiply them by the maximum output values. Example 2 provides an example of the simple computation required.

Given the degrees of membership in the predicate set [crawl = 0.2, walk = 0.4, run = 0.7] and the corresponding velocity values for complete membership to the predicates [crawl = 0.5 km/h, walk = 3.0 km/h, run = 6.0 km/h]. A normalization factor $\boldsymbol{F}$ *computed by taking the sum of each degree of membership, is used to bring the degrees of membership into the range 0.0 to 1.0. The normalized degree of membership is then multiplied by the speed values for the given predicate, yielding a blended velocity* $\boldsymbol{v}$ *that takes into account contributions from every predicate.*

$$\text{ғ} = (0.2 + 0.4 + 0.7) = 1.3$$

$$\boldsymbol{v} = \frac{0.2 * 0.5\ km/h}{\text{f}} + \frac{0.4 * 3.0\ km/h}{\text{f}} + \frac{0.7 * 6\ km/h}{\text{f}} = 4.22\ km/h$$

Example 2 - blending based on membership example

Both *highest membership* and *blending based on membership* are valid approaches to compute the defuzzified values, when to use the various approaches are domain specific.

Fuzzy Logic implements the same binary operators as traditional logic. Table 1 contains the most important logical expressions and their equivalent Fuzzy Equations. Here, *m* corresponds to the degree of membership to the predicate denoted by its subscript.

Logical Expression	**Description**	**Fuzzy Equation.**
¬A	*Negate A*	*1-*m_A
A ∧ B	*A and B*	*min* (m_A, m_B)
A ∨ B	*A or B*	*max(*m_A, m_B*)*
A ⊕ B	*A XOR B*	*min* $(m_A, 1 - m_B)$

Table 1 - Logic expressions in fuzzy logic

A methodology for creating systems for decision making using fuzzy logic is called fuzzy control [15]. The decision system is constructed by combining predicates using logical expressions to form rules such as:

$$A \text{ ^} B \text{ ^} \neg C = X$$

It is important to note that even though the set of variables might entail X, one cannot infer that X is true. Fuzzy logic is used to represent vagueness and thus, one can only know that X is true to some degree from the sentence above. What makes this way of quantifying the world so powerful is that it allows a system to apply a degree of change to its operation depending on the priorities of rules and the outcome of the entire rule set. By querying all rules of the system, the controller can make adequate modifications to all its operations

depending on the current world state. Listing 3 provides an example of how such a rule set could be structured in practice. The rule set could potentially be used to describe the reasoning process of a villager as he is approaching a food supply.

$$full\ AND\ surroundedByStrangers\ THEN\ backDown$$
$$starving\ AND\ surroundedByFriends\ THEN\ fight$$
$$starving\ AND\ surroundedByStrangers\ THEN\ fight$$

Given the following degrees of membership [full = 0.9, starving = 0.1, surroundedByStrangers = 0.4, surroundedByFriends = 0.6], the output values of the above rules will be the following:

$$backDown = min(0.9, 0.4) = 0.4$$
$$fight = min(0.1, 0.4) = 0.1$$
$$fight = min(0.1, 0.6) = 0.1$$

Taking the maximum for each output value, one is left with the following.

$$backDown = max(0.4) = 0.4$$
$$fight = max(0.1, 0.1) = 0.1$$

Listing 3 - Fuzzy rule set example

It should be mentioned that fuzzy logic is not truly a method for uncertain reasoning, but a tool for interpolating a set of crisp input variables in such a way that a set of output variables can partially satisfy fuzzy expressions explaining the expected behavior of a system given said input [15].

When attempting to reason about conflict it is important to have a formal language that allows for degrees of truth such as fuzzy logic. Agents may sometimes be affected in various degrees by different faction memberships, which in turn may not be entirely devoted to some causes. As an example, a faction of teenagers might be opposed to the police, but only to the extent that they would tend to avoid them, but not engage them in direct conflict. Degrees of membership allows for a system where factions interact in a much more complex manner than what would be supported by a “true/false” system.

While certainly useful for games in general, fuzzy logic lends itself well to creating more descriptive worlds, as it is capable of representing degrees of truth much more efficiently than formal logic. It can thus be said that fuzzy logic allows for higher fidelity for representing knowledge of the world as seen by the agent, without introducing complex computational overhead. As this thesis seeks to explore ways to build agents with more detailed knowledge of the world around them, fuzzy logic is a valuable part of the toolbox.

3.7 Semantics for game worlds

Russell mentions in [37] how the "umwelt" of a virtual agent is impoverished. This "umwelt" is the environment that the agent exists in and uses to weight its choices. Russell goes on to discuss how AI researchers through the last decades have refined a narrow skillset, consisting of locomotion and path finding algorithms. An agent observes the world it exists in through navigation nodes and object references. It is rare that two different agents would have a varied understanding of the surrounding world. Russell argues that this narrow understanding of the virtual world makes it much harder for agents to act with authenticity. By embedding more information about the virtual world, and possibly equipping each agent with their own world model, agents will be better prepared to make choices reflecting their personality.

To add more meaning to the environment of a virtual world, objects are given semantics. These pieces of metadata are organized in a graph, allowing them to be traversed by agents searching for meaning. In the *"hunger conflict"* example, an agent could access the semantics of the area storing food in his village, finding that the semantic attribute "edible" is no longer applicable to the food available within (having gone bad). The agent would then access the semantic graph to find other objects in the world that subscribe to the attribute "edible". If the agent was equipped with an inference engine, it would then be capable of determining what object might be the ideal candidate for acquisition to satisfy its "hunger" attribute.

The role of semantics in games is an underexplored topic, but that is coming into its own right in the latest generations of computer games such as "Left 4 Dead" [44]. Semantics in games should also provide information necessary for an agent to determine its current context. If a region had its own semantic set that agents could subscribe to, it could "pull up" information from objects in the region to create a context itself [45]. For instance, if several larger objects have flagged themselves as "dangerous", the area could pull itself up and flag itself as dangerous, allowing civilian agents to reason that it should flee the area while combat trained agents would enter their "escalated conflict state" [22].

While it is important to understand the importance of semantic data for agent reasoning, there are several problems when such information is to be generalized. The amount of semantic information that can be embedded into the world is limitless. Information that might be crucial for one simulator might be irrelevant for another. For example, a farming simulator might have a lot of semantic data concerning soil fertility and composition, while this information might make no sense in a flight simulator. Simply put, this solution would not be feasible both due to the amount of storage space required for this information and the immense performance hit any real time system would take from having to traverse this huge web of information. A solution to this would be to create a system allowing a designer to disable the parts of the semantic system not required by the simulation, as well as adding new relations and descriptors as needed [45].

Semantic modeling allows for an interesting approach to world design. By allowing virtual objects to contain and publish semantics describing services they provide, agents are able to search their world model for providers of the service that satisfy their need. A hungry agent could for example find that two vendors satisfy their requirement. Depending on the current state of the agent, it could also choose a garbage can or stray dog as the service provider.

The idea of using semantics to define objects as service providers is very powerful as it allows for a more flexible design methodology. Instead of having to rewrite parts of the AI reasoning engine, a designer could simply add new methods to the simulation and attach semantics describing the services the object provides. Agents operating within the environment would without modification be able to add these new objects to their reasoning patterns and utilize their services in the same manner as objects created when they were initially designed.

As understanding the situation context is so important for reasoning systems, it makes sense that to create believable scenarios, a great deal of world detail must be made available to the internal AI engine. By creating a system for object semantics, the context of a given situation can be made clearer to the agents attempting to decide their next state transition. By creating a system capable of accepting any semantic set, for then to perform reasoning and formulate plans from these given semantics, one can facilitate the creation of any variation of scenarios. Such a system can arguably provide great enhancement to the levels of reasoning performed by agents in games.

3.8 Neural networks

Neural networks were developed as part of the early work in attempting to create true artificial intelligence. Neural networks were designed to mimic the way our brain is constructed [15], by creating artificial neurons that are combined to form different structures.

While neural networks so far has failed to produce a true working artificial intelligence, they have proved very useful for creating systems capable of learning, as well as producing huge variations in resulting output from small changes to input and weighting of functions inside the network [15]. While this instability has proved a challenge for researches making use of neural networks in learning systems, it could arguably be put to use to generate large variations of gameplay scenarios.

An artificial neuron consists of a set of weighted inputs that are summed together to form the complete input to a threshold function, which returns an activation value [24]. While it is most common for the threshold function to return either 0 or 1 as activation values, it is also possible to have other types of functions. Figure 5 shows the basic layout of a neuron, where weighted input values are summed together before it is sent to the threshold function that computes the activation value that is sent off to the next stage.

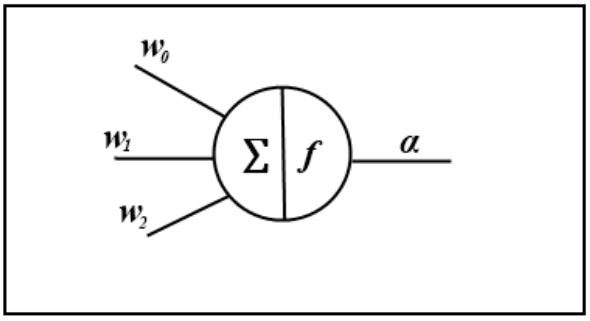


Figure 5 – example of an artificial neuron

Artificial neurons can be structured into different kinds of graphs, which are all referred to as neural networks. It is typical to place neurons in layers, so that one layer computes the weighted values fed into the next stage. The easiest form of neural networks are referred to as perceptrons [15]. Perceptrons are single-layer feed-forward neural networks, which means that the network is composed of a single set of neurons, taking a range of weighted input values and outputting one activation value each, which is interpreted outside the network.

More complex solutions can be formed by adding layers of neurons that use the outgoing activation functions on lower layers as input. To complicate things even further, one can form

cyclic graphs of neurons that exist in adjacent layers to form feedback loops [15]. Such networks are chaotic and almost impossible to predict. However, they can also create some very novel behaviors and effects.

While neural networks will not be used directly in our proposed framework, their potential for creating large variations in output from small variations in input and weighting makes for a good foundation to expand upon.

3.9 Navigation

Finding the best suitable solution to navigating a complex static or dynamic environment is a research field that has been very active for quite some time. In recent years, there has been an increasing focus on fast navigation algorithms that can handle dynamic environments and multiple active agents, such as [46] and [47].

In computer games, it is common to use a graph consisting of waypoints, grid-cells or triangle meshes [48]. These navigation graphs are often made by hand, but recently there have been several successful attempts of automatically generating navigation meshes offline, notably the virtual agents in Valve's source engine [49] and Havok's AI module [7]. The graph is then traversed using a path finding algorithm. In game AI, some variation of the A* algorithm is commonly preferred. Many enhancements and pruning techniques have been proposed for varied scenarios such as the ones listed in [50] and [51].

Most modern game engines offer some form of navigation solutions. However, games of various genres tend to face different challenges. Real time strategy games often seek to render a much larger number of entities on screen at the same time, and thus, the number of queries to the path finding algorithm becomes a bottleneck. Action games and combat simulators render the world in much greater detail and therefore demand that each individual agent navigates the world without showing signs of clipping or colliding with other agents in the scene.

As the framework proposed in this thesis is intended for use in action games, RTS engines are less than optimal. Most commercial game engines like Unreal [52] and CryEngine [53], as well as some open source engines like Panda3D out of Carnegie Mellon come with path finding solutions that work reasonably well for finding paths. However, they tend to require a lot of work to create the navigation graphs. The optimal solution would be to have a framework that works independently of any navigation system; however this could prove

difficult should one consider spatial information as a part of the reasoning process. Therefore, some different path finding solutions should be considered.

3.10 Scaling simulations to larger populations

When developing simulators or games that require real time interactivity, time is always of the essence. While it might be viable to simulate the interaction between a few individuals using a sophisticated mental model [36], doing so for a small city at the same time would not be feasible using hardware available to the average consumer. By using spatial partitioning, this problem can be somewhat circumvented by delegating resources to simulation of agents in the vicinity of the player.

By creating a pool of resources suited to the computational resources available at any given time, resources can be delegated mainly to the agents operating close to the player, leaving agents further away largely inactive. This way the player will enjoy the full extent of the experience created by agents active in the area.

By creating a data structure for storing the mental model of inactive agents, it is possible to take this optimization strategy even further. Kharkar et al. proposes a model where all agents have an internal context that describes their complete mental model and further plan of action [54]. Once the agent that owns the context is no longer in use, all other assets used by the agent is removed, either delegated to another agent closer to the player or removed completely from memory to allow for other resources to be loaded.

A different approach is to only generate the agents that occupy the space around the player. This has often been found to generate less believable results due to the agents' lack of consistency. By generating more advanced plans for the agents as they approach the area of the player, it has proven to be possible to get a more acceptable result however [55]. While the idea of generating extensive plans for agents as they enter the area has merit, it breaks down if one requires agents to be capable of performing tasks and affecting areas outside the player's sphere of influence. It is however a very cost effective approach for simulating "background populations" which is a very important part of realistic simulations [10].

As this thesis explores aspects of simulating crowds in conflict, it is logical that scaling issues must be addressed as one of the core issues. The ideas discussed here will therefore be essential to designing the framework implementation.

3.11 Summary

The purpose of this chapter has been to provide the reader with an understanding of the techniques and principles required to follow the arguments presented throughout this thesis. Thus, the topics presented in the chapter will be essential in the proposed design of the FIF, detailed in Chapter 5. In addition, the reader has been given an overview of normal design techniques for structuring agent reasoning systems, and how navigation is implemented in most games, which has been used to inform the creation of the prototype, discussed in Chapter 7. Finally, a discussion concerning scalability of crowd simulation techniques in games has been provided, for a more complete understanding of the subject matter.

The following chapter will digress from the discussion of AI principles, to present an investigation into available game engines. Chapter 4 presents an overview of several popular game engines and argues for which of the presented engines are best suited for AI research.

Chapter 4 Technologies and frameworks

While Chapter 3 introduced the theoretical foundation required to approach the challenges presented in this thesis, this chapter will address the practical challenges of working with games research. Creating modern games is immensely expensive and time consuming, and it is therefore challenging to create reasonable test scenarios without spending too much time on this alone. A modern game engine provides many tools for quickly implementing common features, which makes it much faster to get a basic prototype working. The different features vary greatly between solutions. This means that one must carefully consider the different options to find one that gives the most benefit for the problem in question. The evaluations prior to building the FIF test scenario is presented here.

To test a high level framework for game AI, one requires some sort of virtual environment. In academic research directed towards AI, the game StarCraft by Blizzard Entertainment (now Activision/Blizzard) has been hugely popular. The RTS AI Research group out of University of Alberta hosts an annual competition where teams from universities across the world compete in creating the best competitive AI [56]. Because of the popularity of RTS games in academic circles, there are many open source solutions such as the SpringRTS engine[57] and the Broodwar API for scripting Star Craft bots [58]. These implementations function out of the box and allow users to quickly implement features within the bounds of the engine.

For action games, however, the author has been unable to find recent games where the AI module is available as open source. Never Winter Nights (NWN) with its Aurora engine has been used for many research projects. While certainly capable of functioning as a development environment, NWN was released in 2002, which makes much of the provided functionality outdated. Another feasible approach is to create a simple game scenario using a game engine or framework. There exists a large amount of proprietary and free to use engines, however the implementation quality and feature set varies greatly from engine to engine.

4.1 Evaluating game engines and frameworks

When evaluating solutions to use as foundation for further work, it was necessary to define a set of evaluation criteria to lead the selection process. As the main goal of this thesis is to investigate the viability of high level AI for use in simulation and games, it stands to reason

that lower level functionality must be available before it can be tested in a reasonable environment. To successfully apply a system for orchestrating agent behavior, the following functionality should be available in the engine:

Path finding is required to make it possible to build any large world. Without a path finding system in the engine, it would either require a lot of effort to implement a reasonably fast pathfinder, or the virtual world would not support any form of complex geometry.

Physics includes locomotion functionality and other actuators for the agents to utilize, as well as collision and so forth. For there to be any kind of gameplay or movement, a physics system is required. This is another component that would require immense work to implement.

An *AI module* including a behavior tree or FSM builder, as well as basic crowd behavior algorithms would make it easier to implement gameplay of the type well suited for the FIF. Such a module would also involve some sort of scripting system or tool for defining behavior models.

Rendering modules are almost always included in game engines and frameworks for game development. As this is one of the most time consuming features to implement properly. For the project in this thesis, no advanced rendering is required, but both GUI and 3D rendering capability is essential.

A *logging system* that provides run time debug functionality without slowing down the game by any major extent is essential for any engine. However, given an engine that provides source code or allows for most game content to be written in a native language, there are plenty of third party solutions to implement this.

Real time editing is a modern feature of many “high end” engines, which basically allows the developer to change the code while the game is running. This allows for quick changes to be applied to game and AI behavior for quick prototyping and testing.

The engine should be either come with source code, or support some form of *high level language*, which allows for the FIF to be integrated with the engine.

Selecting a platform for development is then a matter of finding the one that best supports the listed features. There are several hundred game engines on the market today [59], which makes evaluating all of these an insurmountable task indeed. Instead, a small set of well-established engines with various degrees of conformance to the attributes described above was chosen for evaluation. The main engines and frameworks that were evaluated were

Panda3D, Unreal 3, XNA Framework, Torque, the Havok Framework suite and Unity3D. Table 2 provides an overview of the features available in the different implementations that were evaluated.

4.1.1 Torque 3D

Torque 3D by Garage Games [60] is a game engine that supports full 3D and 2D rendering, with a fully working editor and real time editing functionality. Torque also features a physics engine and has the logging functionality required for development, however there are no AI features available, which means a lot of fundamental work would have to be done before the framework in this thesis could be tested. Torque additionally requires developers to make use of their proprietary scripting language Torque Script, which would lead to any solution developed with it bound to the engine.

4.1.2 Unreal 3

Unreal engine by Epic Games [52] is a high end game engine that supports all of the above features, except Behavior Trees and live changes. Unreal can be used by everyone, free of charge. However, the engine is manipulated using Kismet or Unreal Script. The languages mentioned are both proprietary, which leads to this engine having to be excluded as a potential platform as well.

4.1.3 XNA framework

The XNA Framework by Microsoft [61] provides basic functionality required for interactive media software development. Unlike the engines mentioned earlier, the XNA Framework does not come with a complete rendering engine or editor. However, it does make available all the features required to rapidly create the basic tools required. XNA is written in C#, which makes it possible to integrate third party solutions to satisfy other requirements. While XNA makes it easy to develop the functionality needed, the amount of work required to create a fully functional testing environment makes XNA a relatively risky choice given the time constraint.

4.1.4 Panda3D

Panda3D by the Entertainment Technology Center at Carnegie Mellon University [27] comes with many of the features listed above. It is open source, comes with performance monitoring tools, 3D and 2D rendering and uses Python as its scripting language. On the other hand,

Panda3D does not have a physics and collision package, nor does it directly implement any AI functionality. AI is available in Panda3D using PandAI which is another open source project out of Carnegie Mellon. PandAI offers basic AI functionality such as path finding using Waypoints, obstacle avoidance and basic agent behaviors such as flock, seek and wander. Sadly, Panda3D does not offer physics solution to tie in with their renderer, which would result in additional work to find and implement a viable physics solution for the project.

4.1.5 Havok framework suite

The Havok framework suite developed by Havok.inc [62] contains all the features required for the project. The total set of software solutions include: rendering, navigation, physics, animation, behaviors and LUA script virtual machine support. The suite would probably function as a highly suitable software platform for the faction interaction framework; However, Havok only offers academic licenses on their physics and animation solutions. As Havok does not offer evaluation periods for their software, and the pricing is decided on an individual basis, it was deemed to not be economically viable.

4.1.6 Unity 3D

Unity3D [63] (from this point referred to as Unity) is a modern game engine that focuses on providing game engine functionality to small and midsized development studios. The 26th of December 2011, they released Unity 3.5 which comes with a flexible and optimized navigation suite which includes automatic navigation mesh generation, path finding that supports large groups of NPCs with built in crowd management as well as locomotion systems that tie directly into the physics engine also provided by Unity. To interface with Unity, the developer writes JavaScript (Unity script) or C# with the Mono Framework, an open source implementation of the Microsoft .Net framework. C# with Mono makes it possible to develop and distribute the solution as a stand-alone framework that has no hard dependencies on game engines of frameworks.

4.1.7 Summary

Panda3D (with PandAI), Havok (full suite with component assembly) and Unity both cover most aspects required by the software platform, as can be seen from Table 2. However, Panda3D does not support a physics system out of the box, which Unity does. Unity also supports dynamically linked libraries (DLL) files written using mono or .Net to be utilized

directly with engine components. This allows the faction interaction framework to be developed separately and deployed as a plugin to the test game, which is in accordance with the problem statement. For these reasons, Unity was chosen as the software platform to support development of the Faction Framework.

Table 2 – Features present in the different platforms

Platform	**Path finding**	**Physics**	**AI Module**	**Rendering**	**Logging System**	**Real-Time Editing**	**High Level Language / Non Proprietary**
Panda3D	X	-	X	X	X	X	X
Unreal 3	X	X	X	X	X	-	-
XNA Framework	-	X	-	X	-	-	X
Torque	-	X	-	X	X	X	-
Havok Framework Suite	X	X	X	X	X	-	-
Unity3D	X	X	X	X	X	X	X

4.2 Unity in depth

Having completed the initial feature analysis, Unity was chosen as the engine to use as a platform for further development. While Unity supports most features required, it was important to ensure that the engine would remain stable when simulating larger groups of agents. Unity performance monitoring utilities make it possible to quickly determine the current bottleneck of the target application down to the function call. Using these tools, a

basic 3D world was created and tested with 100, 200 and 400 agents, respectively. These initial tests were run on a MacBook Pro laptop with the hardware specifications listed in Table 3.

Table 3 - Test machine hardware specifications

Component	Model	Details
GPU	NVIDIA GeForce GT330M	Adapter RAM 256 MB
CPU	Intel Core i5 M 520	2 Logical Cores @ 2.40 GHz 3 MB Cache
RAM	Intel i5 Compatible DDR3	4 GB @ 1333 MHz

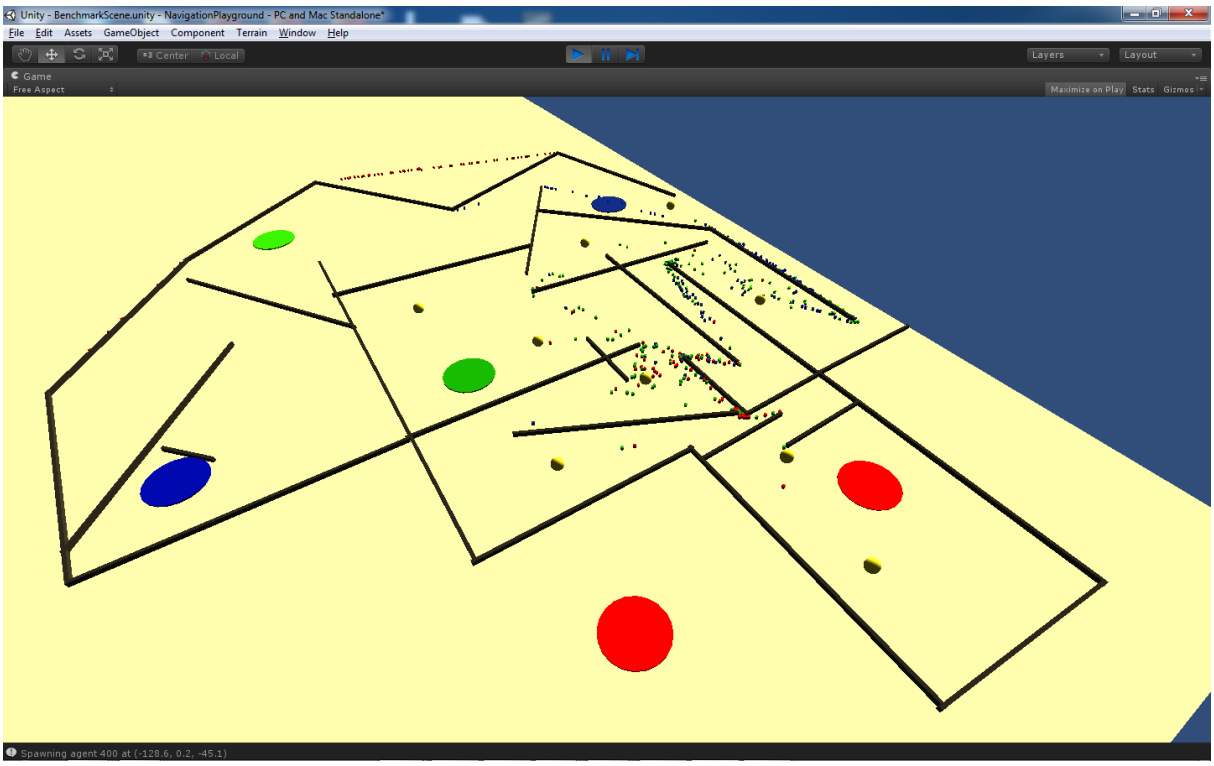


Figure 6 - The scene created for benchmarking Unity for larger groups of virtual agents

The main goal for this simulation was to gather data concerning path finding, physics and

rendering performance. As can be seen in Figure 6, the 3D scenario contains an obstacle course with six locations where agents are generated and placed in the world, marked by the large circles in red, green and blue. An agent is represented by the very small cylinders, bearing the same color as the circle they were generated in. Each agent is given a target when it is generated and randomly picks a new target location (marked by the yellow spheres) once a specified time has elapsed. The time between each target acquisition is randomly set for each agent in a span between 3.0 and 12.0 seconds. This is done to simulate a believable game or simulation scenario, as there are few instances when all agents in a simulation will require a new path at the same time.

The initial tests proved satisfactory, but it was discovered that Unity makes little use of multithreaded processing. This suggests that it should be feasible to simulate a similar number of agents as shown in these tests even with more complex behavior by making use of additional logical cores. Figure 7 contains a graph describing the increase in update time as number of agents increase. As can be seen from the graph, Unity supports more than a hundred agents at a time, while still leaving time for other computations and gameplay. The

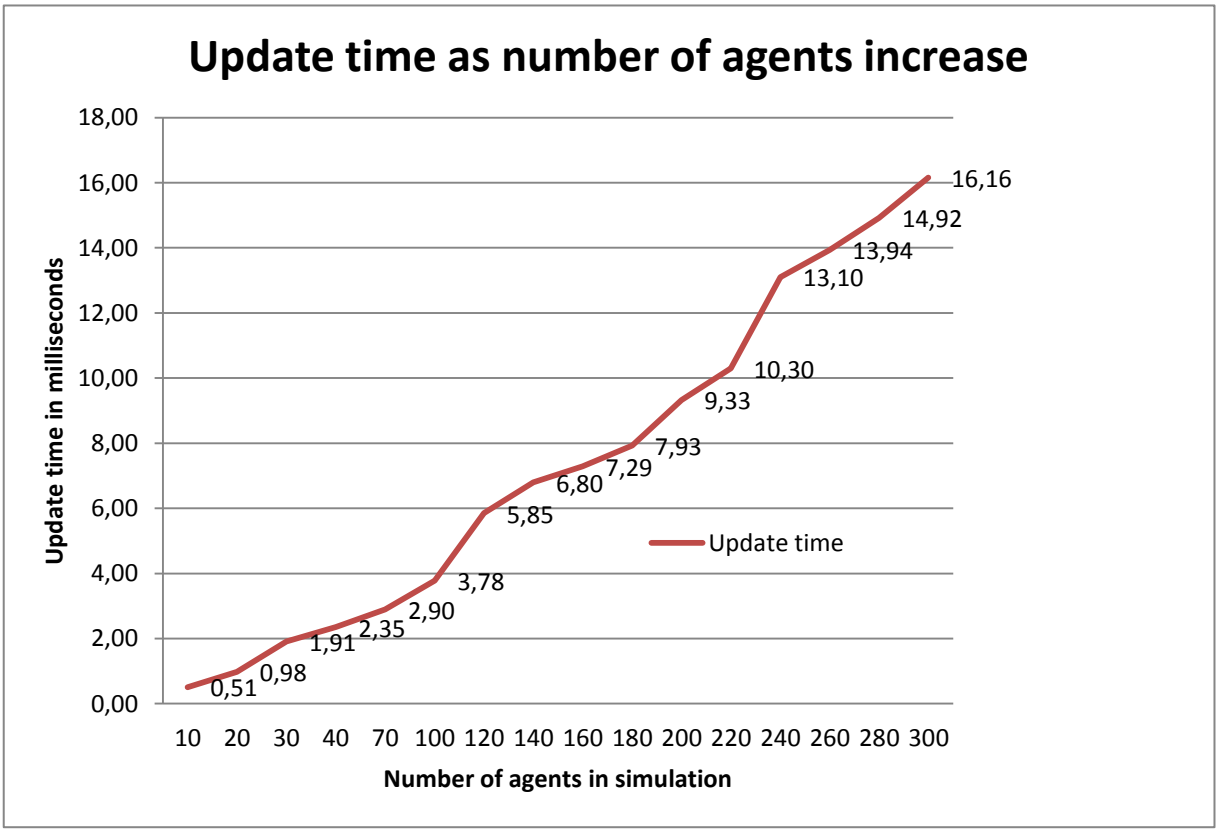


Figure 7 - Changes in update time as number of agents increase

detailed test data and interpretation can be found in each respective chapter on specific Unity

features. This next section will discuss the information shown in figure 7 in more depth, and provide screen captures of the profiling tool used to produce the data.

4.2.1 Unity navigation framework

The unity navigation framework is based on Recast/Detour which is an open source C++ solution for dynamically generating navigation meshes and traversing said meshes [64].

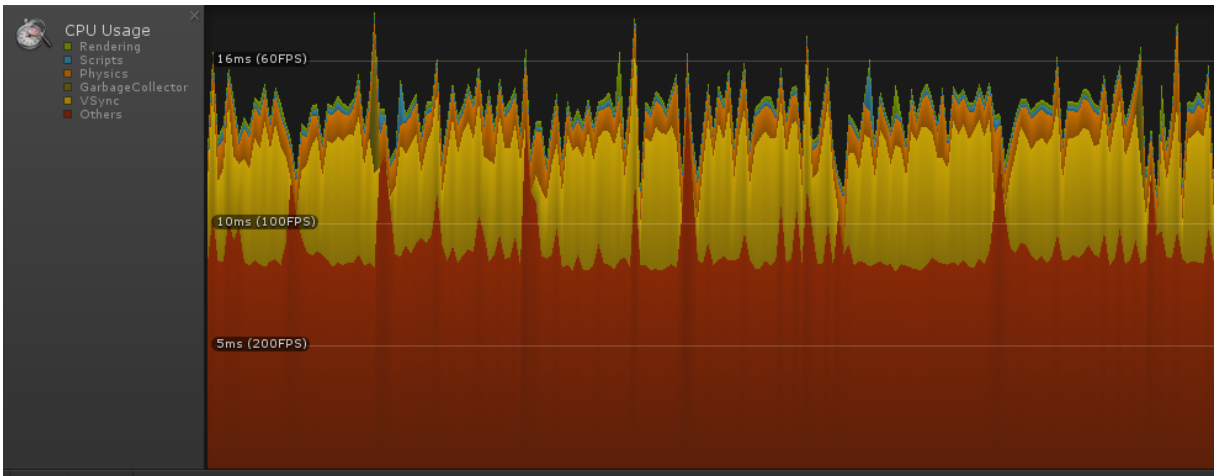
Upon starting the simulation, it would spawn an initial 100 agents. The simulation was confirmed stable, and then, the performance data was recorded before the next batch of agents were spawned and the process repeated. Figures 8-10 are screen captures of the performance tool while monitoring the three different agent numbers. Figure 10 describes the increase in time spent computing crowd avoidance and path finding as the number of agents approach 400.

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
WaitForTargetFPS	69.2%	69.2%	1	0 B	10.83	10.83
CrowdManager.Update	25.7%	2.5%	1	0 B	4.02	0.39
Physics.Simulate	1.7%	1.7%	1	0 B	0.26	0.26
Camera.Render	1.4%	0.1%	1	124 B	0.22	0.01
Overhead	1.1%	1.1%	1	0 B	0.18	0.18
SendMouseEvents.DoSendMouseEvents()	0.4%	0.3%	1	108 B	0.07	0.05
AudioManager.Update	0.0%	0.0%	1	0 B	0.01	0.01
MouseLook.Update()	0.0%	0.0%	1	0 B	0.01	0.01
CameraMovementControl.Update()	0.0%	0.0%	1	0 B	0.00	0.00
GUI.Repaint	0.0%	0.0%	1	0 B	0.00	0.00
HandleUtility.SetViewInfo()	0.0%	0.0%	1	0 B	0.00	0.00
AgentSpawningScript.Update()	0.0%	0.0%	6	0 B	0.00	0.00
Cleanup Unused Cached Data	0.0%	0.0%	1	0 B	0.00	0.00
Loading.UpdatePreloading	0.0%	0.0%	1	0 B	0.00	0.00
ParticleSystem.Update	0.0%	0.0%	1	0 B	0.00	0.00
ProcessRemoteInput	0.0%	0.0%	1	0 B	0.00	0.00
AgentMovementScript.Update()	0.0%	0.0%	100	0 B	0.00	0.00
Graphics.BeginFrame	0.0%	0.0%	1	0 B	0.00	0.00
Physics.UpdateSkinnedCloth	0.0%	0.0%	2	0 B	0.00	0.00
Graphics.PresentAndSync	0.0%	0.0%	1	0 B	0.00	0.00
MeshSkinning.Update	0.0%	0.0%	1	0 B	0.00	0.00

Figure 8 - Performance results for benchmarking scene stabilized at 100 agents

Figures 8-10 provide a large amount of information which is interpreted as follows; the legend describing the different colors in the graph can be read in the upper left corner of each

screen capture, under the title "*CPU Usage*". Each color represents a specific subsystem taking up computation cycles. As the path finding and crowd manager sub systems are new as of Unity3D 3.5 [63], these are listed as a part of *"Others"* in the legend. In the upper part of the figures, left of the legend can be seen compact graphs containing the current frame rate of the application as well as what subsystems are contributing to said frame rate. It is important to note that the benchmark application is running with a cap of 60 frames per second, which means the graph has little information in regards to the highest potential frame rate. On the other hand, it makes it easy to identify what subsystems are demanding the most computation time. The vertical axis of the graph describes the current frame time, which means that a lower value entails higher performance. The horizontal axis describes linear time. The exact point in time has little significance as the simulation was fully stabilized as these graphs were generated, meaning that the patterns depicted were repetitive.

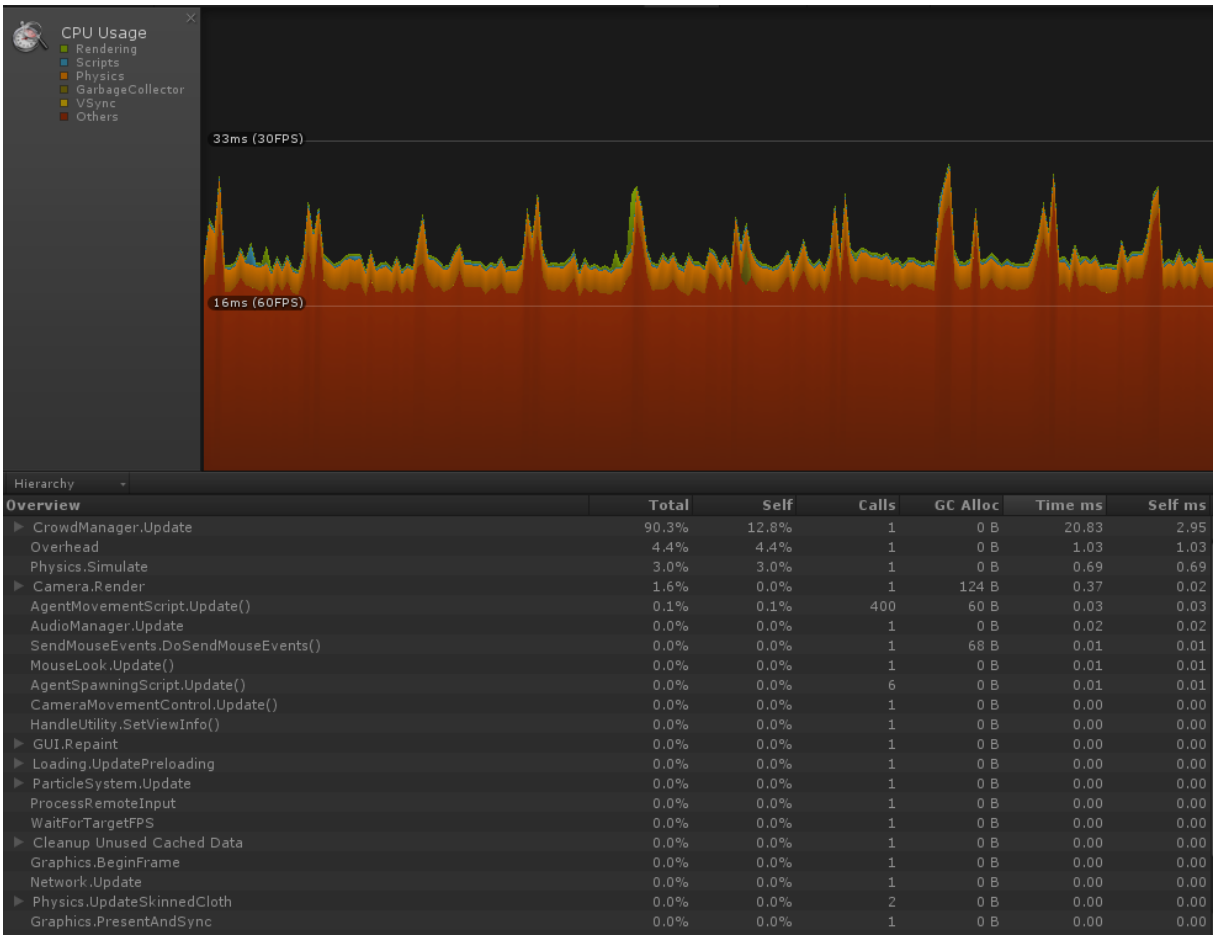


Hierarchy

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
CrowdManager.Update	73.0%	4.6%	1	0 B	11.93	0.76
WaitForTargetFPS	22.8%	22.8%	1	0 B	3.72	3.72
Camera.Render	1.6%	0.1%	1	124 B	0.26	0.01
Overhead	1.3%	1.3%	1	0 B	0.21	0.21
SendMouseEvents.DoSendMouseEvents()	0.5%	0.3%	1	108 B	0.08	0.05
AgentMovementScript.Update()	0.1%	0.1%	200	120 B	0.03	0.03
AudioManager.Update	0.1%	0.1%	1	0 B	0.02	0.02
MouseLook.Update()	0.0%	0.0%	1	0 B	0.01	0.01
AgentSpawningScript.Update()	0.0%	0.0%	6	0 B	0.01	0.01
CameraMovementControl.Update()	0.0%	0.0%	1	0 B	0.00	0.00
GUI.Repaint	0.0%	0.0%	1	0 B	0.00	0.00
HandleUtility.SetViewInfo()	0.0%	0.0%	1	0 B	0.00	0.00
Cleanup Unused Cached Data	0.0%	0.0%	1	0 B	0.00	0.00
Loading.UpdatePreloading	0.0%	0.0%	1	0 B	0.00	0.00
ParticleSystem.Update	0.0%	0.0%	1	0 B	0.00	0.00
ProcessRemoteInput	0.0%	0.0%	1	0 B	0.00	0.00
Network.Update	0.0%	0.0%	1	0 B	0.00	0.00
Physics.Interpolation	0.0%	0.0%	1	0 B	0.00	0.00
Physics.UpdateSkinnedCloth	0.0%	0.0%	2	0 B	0.00	0.00
Waypoint.Update()	0.0%	0.0%	9	0 B	0.00	0.00
Graphics.BeginFrame	0.0%	0.0%	1	0 B	0.00	0.00

Figure 9 - Performance results for benchmarking scene stabilized at 200 agents

For Figure 8 this is mostly "*VSync*" which is the system responsible for keeping the application at 60 frames per second. In other words: The application spends most of its time idle. More detailed information concerning the computation time of each sub system can be found in the lower part of each figure. In the lower left is the name of the function or sub system. To the right is the *"Total"* which describes the percentage of one frame that is spent performing the listed task. *"Time ms"* provides the same information, but in milliseconds. The important information to be gleaned from these values is that the CrowdManager sub system, which is responsible for managing groupings of virtual agents, is the only system that seems to be heavily affected by the increase in agent population.



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
CrowdManager.Update	90.3%	12.8%	1	0 B	20.83	2.95
Overhead	4.4%	4.4%	1	0 B	1.03	1.03
Physics.Simulate	3.0%	3.0%	1	0 B	0.69	0.69
Camera.Render	1.6%	0.0%	1	124 B	0.37	0.02
AgentMovementScript.Update()	0.1%	0.1%	400	60 B	0.03	0.03
AudioManager.Update	0.0%	0.0%	1	0 B	0.02	0.02
SendMouseEvents.DoSendMouseEvents()	0.0%	0.0%	1	68 B	0.01	0.01
MouseLook.Update()	0.0%	0.0%	1	0 B	0.01	0.01
AgentSpawningScript.Update()	0.0%	0.0%	6	0 B	0.01	0.01
CameraMovementControl.Update()	0.0%	0.0%	1	0 B	0.00	0.00
HandleUtility.SetViewInfo()	0.0%	0.0%	1	0 B	0.00	0.00
GUI.Repaint	0.0%	0.0%	1	0 B	0.00	0.00
Loading.UpdatePreloading	0.0%	0.0%	1	0 B	0.00	0.00
ParticleSystem.Update	0.0%	0.0%	1	0 B	0.00	0.00
ProcessRemoteInput	0.0%	0.0%	1	0 B	0.00	0.00
WaitForTargetFPS	0.0%	0.0%	1	0 B	0.00	0.00
Cleanup Unused Cached Data	0.0%	0.0%	1	0 B	0.00	0.00
Graphics.BeginFrame	0.0%	0.0%	1	0 B	0.00	0.00
Network.Update	0.0%	0.0%	1	0 B	0.00	0.00
Physics.UpdateSkinnedCloth	0.0%	0.0%	2	0 B	0.00	0.00
Graphics.PresentAndSync	0.0%	0.0%	1	0 B	0.00	0.00

Figure 10 - Performance results for benchmarking scene stabilized at 400 agents

With 100 active agents in the scene, the Unity Crowd Manager, which handles agents navigating the scene and avoiding other agents, takes approximately 4 milliseconds (ms) per frame to run, which can be seen in Figure 8. With 200 agents it averages at 11.8 ms, as shown in Figure 9. 400 agents have the crowd manager running at 20.83 ms when stabilized, as can be seen in Figure 10. Figure 11 provides a summary of the key data from figures 8-10. This figure shows the time spent updating the crowd manager and physics subsystems of unity,

with 100, 200 and 400 agents respectively. The time is listed in milliseconds. It should be mentioned that an acceptable frame update time can vary greatly, depending on the simulation type, however most game engines default to 30 or 60 iterations per second [28, 65], for the entire frame, which includes rendering the scene. This translates to an average time per frame of 33,333 or 16,666 milliseconds respectively.

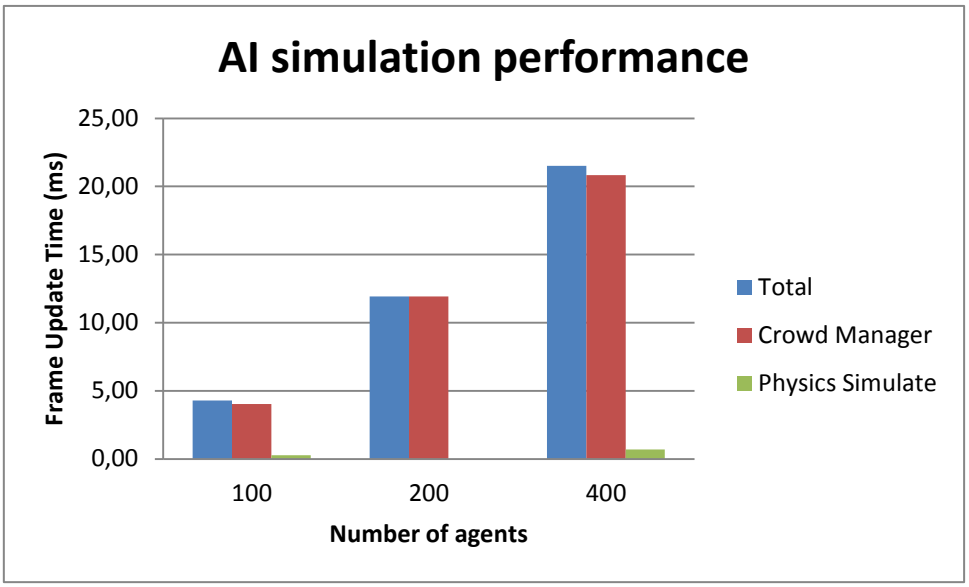


Figure 11 - Profiler Summary

Simulating 400 agents in real time is more than adequate to create the test case as it is described in the introduction, and it can thus be concluded that Unity is a good choice for testing the faction interaction framework.

4.2.2 Unity editor and script development

The unity editor is an all-inclusive world editor that makes it fairly easy to create complex worlds that can be navigated by virtual agents. The editor allows for defining component based game objects in a drag and drop fashion, as well as full scale code solutions to be deployed and run on top of the game. By using Unity together with the Faction Interaction framework one will have a development scenario that is arguably very similar to that of an average game development studio. Figure 12 contains a screen capture of the Unity3D editor. The drag and drop functionality that is automatically available for all scripts created in Unity can be seen on the right hand side. The two scripts attached to the "GreenWarrior1" agent are "Nav Mesh Agent" and "Hunger Games Agent", responsible for interacting with Unity navigation and the FIF respectively. In the upper left is the game scene, the two villages used in this simulation are represented by the 6 black boxes that represent different houses. Below

this is an overview of all assets in the scene, under “Hierarchy” and all the content available, which is listed under the “Project” tab.

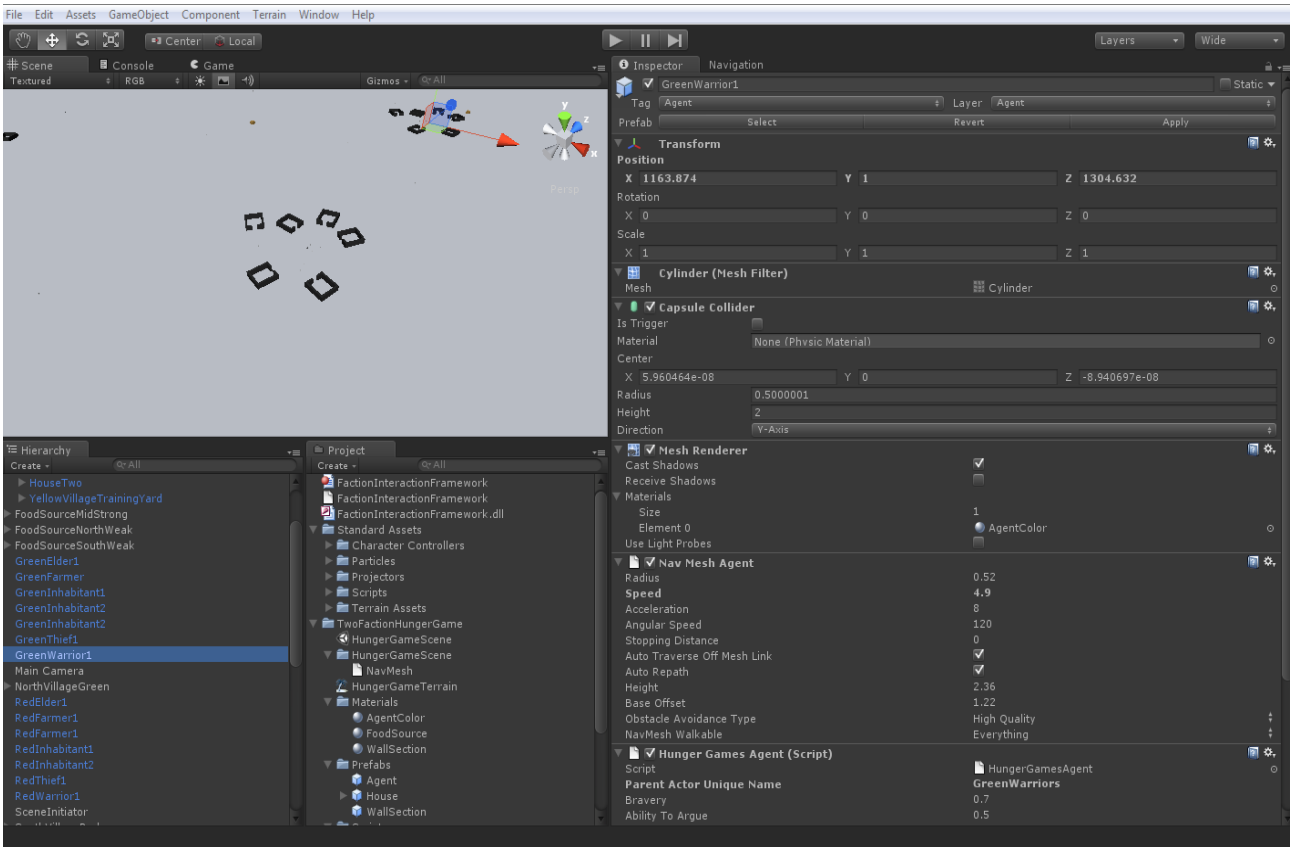


Figure 12 - Unity3D editor

4.3 Summary

This chapter has presented an evaluation of the feature set available in a small subset of the game engines and frameworks available on the market today. Concluding that Unity3D appeared to be the best choice for use in prototyping the FIF, the performance characteristics of the engine was evaluated.

Several of the essential features of Unity were discussed in depth, and a test of the AI, path finding and physics systems was performed with 100, 200 and 400 entities. The results led to the conclusion that Unity was capable of meeting the needs for the FIF integration and Scenario prototype. Chapter 3 presented the advanced theory of AI, while this chapter described the practical tools required to create a prototype scenario, providing the foundation required for the theoretical design of the FIF, which is presented in Chapter 5.

Chapter 5 Core design

Based on the knowledge and selected components from Chapter 3 and 4, we describe and discuss, in this chapter, the core feature design for the FIF. The faction interaction framework is a framework for allowing complex interaction between agents in a virtual world to create gameplay through emergent conflict. The FIF allows game programmers and designers to define the parameters for their virtual world, and then leave some of the gameplay generation to the FIF, potentially reducing cost or increasing replayability.

The concept of the FIF is to allow for a more dynamic approach to game and simulation development. Instead of creating strict rules for how each scenario occurs in a given succession, the developers define the parameters for their world, and let the FIF do the rest. In other words: developers define the creatures that inhabit the world they wish to create. They then define the relationship between the creatures and in what ways they can interact with each other. Finally, they define the world in terms of desire values; "*what do the creatures in this world want? How do they get it?"* Given these parameters, the FIF will then attempt to simulate the interaction between the various entities in the world, and suggest new actions for the agents as parameters change over time.

To accomplish this, the FIF must have a system for representing the world in the terms stated above, as well as a reasoning engine to drive the simulation forward. An abstract representation of actions applied to objects, and actions applied to other agents, must exist, for these to be suggested to the agents in the world. These ideas and concepts form the requirements of the core FIF feature set.

The core features of the FIF deal with concepts required to generate gameplay through conflict generation. Figure 13 provides an overview of the design discussed in this chapter. Note how the faction interaction framework is mostly isolated. The only direct relations required between the FIF and the game world is smart object information in objects worthy of conflict, as well as ties between actors and agents. The arrows in Figure 13 describe dependencies between elements.

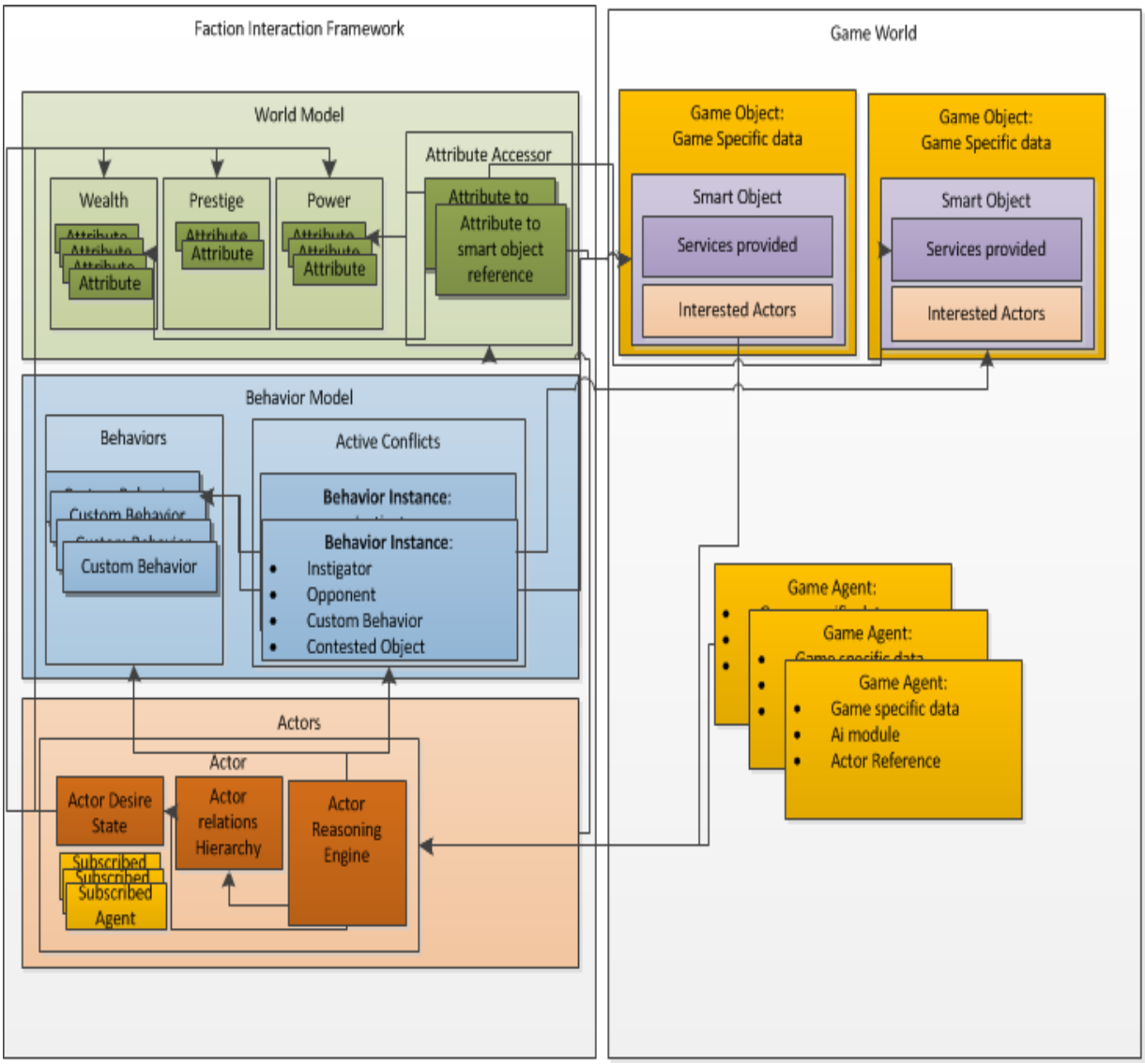


Figure 13 - Design overview

The core design description is split up into several parts, as is evident from Figure 13. Section 5.1 describes how the game world is made accessible to the FIF through embedding game objects with semantic information and attaching smart object structures. Section 5.2 provides an overview of the behavior model design. The behavior model is the "end result" of the framework, as this is where conflict resolution strategies are selected and suggested to the game engine. Section 5.3 then explains the world model. Unlike the world description, which deals with describing the game world to the FIF, the world model is the generic world representation internal to the FIF. Actors and how they choose when to initiate conflicts and when to request services is described in section 5.4Actors, while 5.5 introduces an interface for how agents in a game receives action suggestions from the actors. Finally, the internal details of the actor reasoning engine are laid out in section 5.6.

5.1 World description

This part of the core feature set deals with designing how a game world already created can be represented in the FIF. Going back to Figure 13, the following section, and its subsections, deal with the part of the FIF design that is integrated with the virtual world. Sections 5.1.1 and 5.1.2 deal with how one describes the "look and feel" of the game world, with smart objects and semantic information. In Figure 13, this is the part contained within game objects of the virtual world. Sections 5.1.3 and 5.1.4 deal with designing how the FIF can represent possible interactions that the game world allows through actions and services. These are also contained within the smart object structure, and complete the integration between virtual world and the FIF. This is described further in section 5.1.5. Combined, these sections form the basis for the FIF to understand what part of the game world is to be taken into account when conflict is to be generated.

5.1.1 Smart objects and semantic information

As mentioned in section 3.7, if one was to implement all possible semantics about all objects in the virtual world, the resulting dataset would be too large. Because of this problem, a more viable approach is to create a system that allows designers to define semantics and how they relate to their game world instead of forcing them to choose from a predefined set. Kessing et al. combine smart objects and semantics in [66] to demonstrate a game world where all artifacts in the world are described through a combination of classes of objects, physical attributes, services the objects offer and actions that affect them. This level of abstraction lends itself well to the FIF implementation as it will allow for any kind of world to be defined.

Classes are quite simply definitions of objects, such as a "vehicle" or "person". Classes can be part of an inheritance hierarchy, for instance, the "car" class can be a child of a "vehicle" class. While this classification solution might be important for an agent that uses an inference engine to reach conclusions about the world, it is not necessary. Kessing et al. describe classes as *"a collection of entities based on their essential common attributes"*. For simplicity, defining hierarchies of classes will not be implemented in the framework. Classes are simply the definition of a collection of attributes that describe some entity in the game world.

5.1.2 Attributes

Attributes characterize an entity in the virtual world. A typical attribute for an apple would be "edible" or "dangerous" and have some descriptive value assigned to it. Kessing et al. describe how simply having this "tag" is not sufficient, and that each attribute must have a corresponding value to make it possible to use them for reasoning. Each entity subscribing to an attribute will create a new entry in the attribute map containing one floating point value that describes the entity's membership to that attribute. Using fuzzy logic, it is up to the user of the framework to define the defuzzification algorithm that makes sense of the value. For example, "edible" may use a Boolean function to describe its membership, while a "weight" attribute would use a linear or exponential function, depending on the context of the game. When an agent seeks to reason about the world, it can query the semantic data structure for a list of all objects that have a specific attribute. While it can be argued that the storage and querying of the semantic data should offer more advanced functionality such as those offered by query languages, these features are beyond the scope of this thesis.

Through the use of custom defuzzifers, designers making use of the framework can use attributes to describe virtually any type of information about their world that is required to inform the agents that inhabit it. Combining this with the *attribute accessor,* one can create a web of information that any agent (or player) can make use of to further reason about the world around them.

Attributes are complimented by an *attribute accessor* system. By forcing all attribute assignment to go through the accessor before they can be assigned to an object, a web of information is defined in the accessor. The basic functionality that is used for demonstrating the use of this design is simply to allow the user to query the accessor for any object that has this attribute. This way, any agent interested in knowing more about the world around it, can query the attribute accessor to find objects with this type. As an example of this functionality, a pickaxe has the attribute "weapon" attached to it. Should an actor decide it wishes to optimize its "battle ready" attribute, it could then use the accessor to find any objects in the world that has this attribute attached to it.

This implementation allows the designer to describe the virtual world with as many details as is deemed appropriate for the implementation. Attributes and the accessor system can also be used without the world model, to simply provide a means for the designer to describe the world they are working with. The way attributes are stored within the FIF and their relation

to the game world is shown in Figure 13. Here one can see how the different attribute groupings stored within the FIF, and how the game objects are described to the FIF using said attributes, through the use of smart objects.

5.1.3 Actions

Actions describe the process performed by an entity, yielding some attribute change or through generating new entities of some kind. Actions can be performed by an agent in the system, or as the end result of a service. A typical example of an action could be "open" and "reduce hunger". While "open" could be part of some service activation requirement, "reduce hunger" would be the action applied to an agent or other smart object as an end result once all the specifications in the smart object has been fulfilled.

An *action* contains a list of attribute modifiers which defines the changes activating the action will have on the agent and the world around it. It contains temporal information that describes how much time it takes before the action is activated. The temporal information is not acted upon by the implementation itself, as the implementation should be able to handle any scenario, with arbitrary time representations. It should be noted, however, that it is assumed that all temporal information is of the same timescale, as the number is used in raw form for calculations regarding yield over time as described section 5.6 which describes the actor reasoning engine.

Actions can be used both as a way to explain the outcome of some triggered event, but also as a descriptor for the game to understand what simulations to trigger in the virtual world. To facilitate this behavior, all actions have an "activated event" that can be listened to. This event is fired when the access begins. This way, some simulation manager can initiate animation cycles, sound effects, particle effects or other actuator functions. For instance, an agent performing the "open" action on a box of aid food would trigger an animation on its avatar, playing an interaction animation, while the box would also trigger an animation to open.

5.1.4 Services

Services are more complex in that they describe an approach of how to obtain whatever the artifact offers. A service is described as *"an entity's capacity to perform an action, possibly subject to some requirement"*. A service thus describes how an attribute can be modified by the use of it, for instance "buy chocolate" is a service that perform the "satisfy hunger" action which in turn reduces the value of the "hunger" attribute of the agent that chooses to interact

with it. While Goncalves et al. [43] describe a more sophisticated approach on how to describe the interaction process to such an extent as to what animations to play and how to modify them to model grasping behaviors, such an implementation would be too complex for this proof of concept. Instead, a service will be defined in a simplified version of the one presented by Kessing et al.

A service can have *attribute constraints*, for instance, the service "provide food" offered by a cornfield might have the attribute constraint of *"ripe>0.9"* which would entail that the field would have to be more than 0.9 ripe to be usable. How this information is interpreted is arbitrary and left to the designer making use of the system. A service can also have *action constraints* which specify certain actions that must be performed by the agent for the service to activate. In the case of the food provided by the cornfield, an action constraint could be "*harvest*". Third, a service has a *temporal property*, which specifies how long the service will take to complete. Finally, a service has a *spatial property* which describes the area affected by the activation of the agents that will be affected by the service. Kessing et al. also suggest having an interaction requirement prioritization that specifies in what order the requirements must be filled for the service to function as intended, this can simply be represented by a list structure, allowing designers to choose if they wish to honor the ordering or ignore it.

5.1.5 World description integration

The following design is proposed as a result of the features described above: A smart object is defined by a collection of attributes with corresponding floating point values that are stored for each instance of the attribute. These values describe the degree of membership to an arbitrary predicate set for each attribute, depending on a user defined membership function. In addition, these objects may define a set of actions they may perform through invocation from the AI system, as well as services said object can provide. Smart objects are embedded as parts of the game object and used exclusively by the AI system to reason about the world. As actions are not explicitly defined, they can be defined by the designer and utilized to manipulate both the AI and the virtual world in all ways required to move the game world in the correct direction. The basic design layout of the smart objects implementing semantic attributes can be seen in Figure 14. The figure also shows how this relates to a typical game object using a composition based object structuring. Here, the smart object is composed of an arbitrary amount of semantic attributes that describe the object itself. These attributes are not used by the FIF, but can be used by the game designers to allow for special considerations to

be taken regarding objects with specific attributes. A smart object is also composed of an arbitrary set of services, which contain actions in the form of constraints and one result action. These are again described by the modifiers they apply to the world around them.

This implementation is highly abstract, allowing for practically any world to be described, using these basic building blocks. In addition, as expanding the "umwelt" of the AI is simply a matter of adding another attribute to objects in the world, the level of detail (LOD) can easily be adapted to the current need of the solution. As the current design proposes that all attributes, and where they are attached, should be registered in the *attribute accessor*, it is easy to see that storage requirements and graph traversal time will increase rapidly as additional detail is added. This problem can be handled by implementing a data management system such as a relational database or similar [67].

It is important to note that not all artifacts that exist in the virtual world need necessarily be "smart". Objects in the game that have no value to actors do not benefit from attribute data, instead these objects are simply interpreted by the AI system as collision obstacles through the pathfinder and ignored in other respects. Only objects that possess some form of resource property or that the AI system should be able to use for planning are embedded with smart object features. While this might initially appear as limiting the interactivity of the solution, it is important to consider the combinatory explosion that could result from embedding too many semantic attributes in all objects in the world. A novel approach to enhancing this implementation would be to support large quantities of possible attributes in the engine and allow designers to disable or enable hierarchies of classes and attributes as they are deemed useful for the simulation as proposed by Tutenel et al. [45].

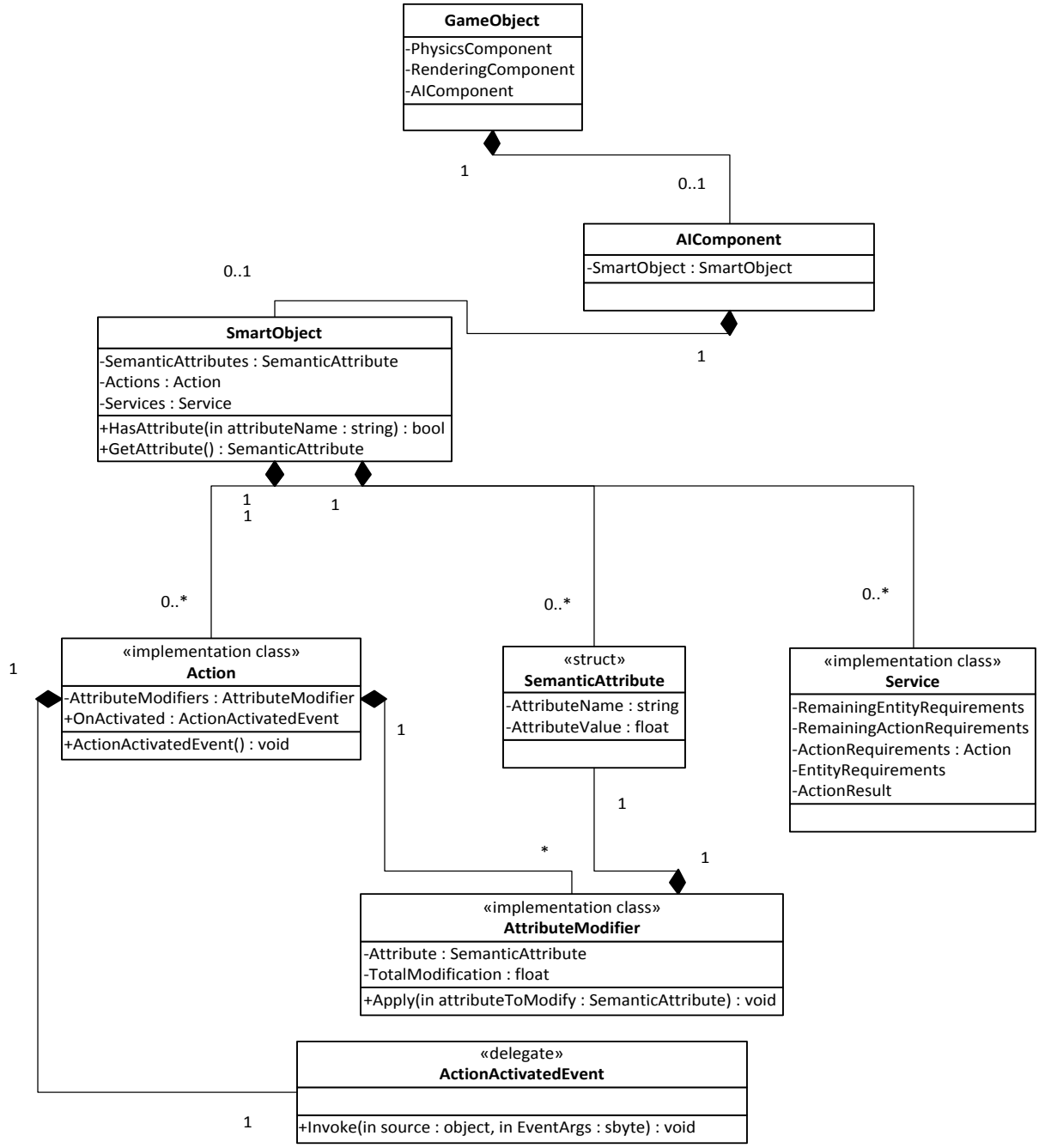


Figure 14 - Smart object design

5.2 Behavior model

The Behavior model defines a set of conflict resolution strategies available to actors in the simulation. The realization of a behavior is defined by the developer making use of the framework. In other words, the FIF will trigger a behavior to resolve a conflict between factions. However, the way this conflict resolution is carried out is left to the developer. A behavior instance consists of a value describing the faction relation space required for the behavior to be valid, the wealth, power and prestige space required to execute the behavior, as well as a list of custom attributes that are required to activate said behavior. The behavior to activate is selected through evaluating all possible behaviors and selecting the one that has the smallest total difference when compared to the state of the actor, multiplied by the current hostility towards the contesting actor. In addition, custom attributes are used as filters, which mean that a behavior containing custom attributes can only be activated by an actor that possesses all the custom attributes listed in the behavior. Once the behaviors have been filtered out, custom attributes are used as part of the difference sum. The process is described in the pseudo code in Listing 4, with function descriptors listed in Table 4.

```
currentBestFitBehavior = null;
bestTotalDifference = float.MaxValue;
for each behavior in behaviorModel.Behaviors

    if actor does not have custom attributes required by behavior

        continue;

    endif

    calculatedTotalDifference = actor.CalculateTotalDifference(behavior);
    calculatedTotalDifference *= GetRelation(actor, contestingActor);

    if calculatedTotalDifference < bestTotalDifference
   OR currentBestFitBehavior is null
        currentBestFitBehavior= behavior;
        bestTotalDifference = calculatedTotalDifference;
  endif
endfor
```

Listing 4 – Behavior evaluation pseduo code

Function Name	Function Description
GetRelation *in:* *actor : Actor, contestingActor : actor* *out: relationRating : float*	Looks up the relation matrix for the two actors and returns the relation the querying actor has to the contesting actor. This value ranged from 0 to 1.
CalculateTotalDifference *in: behavior : Behavior* *out: totalDifference : float*	Calculates the difference between each element in the behavior and the corresponding state variable in the actor. The variables made a part of the result are: wealth, prestige, power, and any custom attributes required by the behavior.

Table 4 - Behavior evaluation function descriptors

Once the actor has selected what behavior to activate, a behavior mobilization object is instantiated. The behavior mobilization object contains a reference to the behavior that was instantiated, the smart object at the center of the conflict, the instigator and its target, as well as the current conflict state. The conflict state is a single precision floating point variable ranging from 0 to 1, where 0 signals complete failure for the actor and 1 signals success. The behavior mobilization instance is kept in the mobilizations list as long as the variable is not set to 0. An actor will only attempt to utilize services on smart objects that have a behavior state value of 1 attached to it, or that does not have contestants. This way, a conflict is automatically generated each time more than one faction attempts to make use of the same smart object. However depending on the factions' affiliations with each other, there will be different conflict strategies employed to share these resources. It is up to the developer making use of the framework to modify the conflict state of each mobilization instance. However, the faction interaction framework monitors the variable and makes changes

according to the new value. Actors that are members of the same faction may choose to join the same mobilization, if they also find that said mobilization is the correct approach. An actor of the same faction may also activate a new mobilization of a different type should its reasoning engine conclude that another approach is more viable in accordance to its internal state.

The act of initiating, joining and resolving conflicts is the main goal of the FIF. All other parts of the framework are designed to allow for this behavior to take place. It is through conflict that interesting gameplay is generated. As all other components of this design, no gameplay is actually created by the FIF. The FIF simply recommends possible ways to stage a new gameplay scenario, which can be ignored or made use of, by the game that implements the framework.

5.3 World model

The world model is based closely on the implementation proposed by Medler et al [22]. In the FIF, the model is a global set of metadata that is accessible to all agents and actors operating within the world. The world model is divided into several data sets, describing different aspects of the world.

5.3.1 Desires

Desires are set of semantic attributes that describe how they are relevant in the world. These attributes are defined by the user and put into three different categories: Wealth, power and prestige. Each attribute is defined by a data structure containing the attribute name as described in section Smart objects and semantic information, as well as a weight function, ranging from 0 to 1, that describe how important this attribute is in comparison to other attributes of the same category as the membership value changes over time. These values are then used by factions to decide what attribute to improve on next.

5.3.2 Organization

Organization contains faction meta-data that describes the different factions of the world and their relation to each other. Once all factions have been added to the organization model, a matrix containing the entirety of relations between factions is generated. The default state of all factions is neutral relations; this can be changed by the user once the matrix has been generated. The matrix is dynamically updated by the faction controllers as their relation to

other factions change. These interactions are described in section Behavior model. There are additional features to the organization aspect of the world model described by Medler et al. [22]. In their paper, organization also describes internal structures for a faction and rules that define how an agent may leave its faction. In the FIF, these features are not implemented for the sake of simplicity, but are discussed further in the section on further work.

5.3.3 Mobilization

Mobilization contains data on possible areas that can be used for staging escalated conflict scenarios and what form of interaction is required by the other faction for the conflict resolution strategy to define an outcome. In other words, the mobilization data structure can be visualized as a *"notice board*" used by factions to decide when and where to meet to resolve their conflict.

In addition to the features described above, there is a final world model aspect in the system proposed by Medler et al. *Solidarity* defines rules for how and when an agent may break with its parent actor, be that a group, faction, or a belief system the agent subscribes to. Breaking with the parent actor can be to simply refuse to participate in some conflict resolution strategy employed by the parent actor, or to leave the actor outright. While this aspect of the world model offers many novel applications, especially for simulations where interaction with agitated populations is the main focus, it was not implemented in the faction framework to avoid further complicating use of the framework when applying it to a game design. The further work section describes how the solidarity aspect would affect both the implementation in itself and potential benefits that could be gleaned from the feature.

The World model is the core of our framework, and is used by all systems to decide what attributes to improve upon and how to affect the world to further the agendas of the actors in the system. Because of this, the entire framework is designed around the concept of a world. The world object is created through incrementally adding all desires, factions and mobilization areas that should be present in the world on startup. Only objects that have been registered with the world object are available to the reasoning engines used by Actors and agents.

5.4 Actors

The system responsible for parsing and treating the data provided by the faction framework is called the actor. An actor is an entity that attempts to exert its will upon other entities and the world to maximize its total desire satisfaction. An actor can be a single agent acting on its own, or a large organization with many sub actors attached to it.

Each actor has its own set of desires, derived from the world model, in addition to any other desires attached to it by a designer. Unless otherwise specified, an actor will perform autonomously utilize behaviors, to maximize its desires satisfaction. An actor that has other actors attached to it may request that behaviors are performed by one of the attached actors instead. Attached actors will immediately cancel the behavior they were currently executing and attempt to accomplish the behavior defined by the parent actor.

Each actor contains a *smart object* structure, as described earlier. This structure, while initially empty, can be initialized to allow other actors to reason about the actor. By using the smart object structure to define an actor with services, actions and attributes, actors can form strategies surrounding individuals as well as artifacts. As an example, consider a world model where global enterprises use any means necessary to get ahead. A faction, faction “blue”, is currently at a high level of conflict with faction “red”. Because of the high conflict rating, “blue” faction infers that its highest relative increase in utility when compared to red is by reducing “red” faction’s knowledge value. It searches through all actors in the system with the “red” attribute that also offer the generate knowledge service, and attempts to apply a conflict resolution strategy that apply to the knowledge value.

All actors are embedded with a *behavior model* that contains all the conflict resolution strategies the actor may employ. These are, as all conflict resolvers, defined by the designer for the specific game and contain meta-data describing the potential output of the function. As described earlier, the actor will employ any behavior as requested by the faction system. While Medler et al. [22] suggests that actors should also reason on whether to accept these orders or to disregard them, it can be argued that this would greatly increase the work required to implement specialized behavior at the faction level, as well as introduce unwanted complexity to the framework. Actors may also use the behavior model to resolve conflicts between them and other actors, groups and factions. However, it will automatically end whatever it was doing to perform a task assigned to it by the faction to which it is a member of. It is important to note that while the actor chooses what behavior model to employ, the

choice of executing the behavior is passed on to a sub actor and finally to the agents attached to the actor. It is, in the end, up to the agents of the world to exert the will of the actors, and because the faction interaction framework seeks to define an abstract approach applicable to a multitude of scenarios, the way these behaviors are executed in the world must be left to those making use of the framework.

The actor holds a reference to the *world model* as described earlier. The world model is a global data set that uses semantic data to define the attributes relevant to the world in which the simulation takes place. In other words, the world model contains attribute tags relevant to wealth, power and prestige. In addition, each actor can be equipped with a local data set that describes attributes relevant to the specific actor. These attributes also require a defuzzification function that describes how the attributes affect the actor's wealth, power and prestige values.

Finally, the actor contains a *reasoning engine* which decides what attributes should be attended to next and queries the smart object system for information on how to obtain these resources. It should be noted that the actor in itself has no effect on the world around it, it may only request that the game performs the actions it suggests, it is up to the designer if the actor is indulged through manipulating actuators in the game object or not. The way the actor knows that the action has been performed is by monitoring the value it attempted to change and seeing if it changes for the better. If nothing changes, the reasoning engine will continue to improve upon this value until something else takes priority.

5.5 Agents

Agents in FIF are simply a term used to generally describe any entity that is interested in listening to events related to conflict resolution. While the FIF in itself only considers the information present within the framework, the agent is the system that is to be determined specifically by the designer. The agent is an abstract definition of functionality that is required by the framework to interact with the world. The main functionality required by the framework is that the agent is able to respond to behavior requests as well as capability of spatial translation.

How these capabilities are implemented is left to the designer making use of the framework. In truth, a system without any agents, simply relaying attribute changes to the actor model

would work just as well for the FIF, but it would make little sense to create a system that has no forms of output.

5.5.1 Behavior requests

A behavior request is the main way the framework resolves conflicts with other actors in the world. The actor, which is responsible for the behavior of the agent, chooses a behavior it wishes to see executed. This behavior request is then sent to the abstract function of the agent in question, which in turn chooses to execute or disregard this request. How the execution of the behavior request is implemented is left to the user of the framework, as it is inherently scenario specific. The actor responsible will assume that the behavior is being executed and only send a new request once the desire states of the actor changes enough for there to be another, more important desire to consider. A behavior request is issued through passing a pointer to the behavior to be executed and a list of actors affected by the behavior.

5.5.2 Action requests

Action requests are used to interact with the world in a way that will modify the behavior values of the actor. Through these requests, the actor can issue an order for an agent to perform an action on a specific entity capable of accepting the given action. Like behavior requests, action requests are assumed to be performed by the agent in the best way possible given the current conditions of the world. The action request is invoked by an actor by passing a smart object reference, an action reference and an object pointer containing the entity that the smart object data is attached to.

As with all other output from the FIF, requests do not directly impact the game world. Requests are recommendations relayed to any listening agents, regarding how best to optimize the world for the actor the agents are subscribed to.

5.5.3 Summary

All simulations that seek to utilize the FIF must have their virtual agents implement the behavior and action requests for the framework to function to its full extent. It is through these interfaces actors seek to interact with the world they are put in. As the faction interaction framework has no prior knowledge of the scenario designers seek to create, this implementation must be left to those making use of the framework. The only assumption

made by the framework, as mentioned earlier, is that the world inhabited has an understanding of time, and therefore make requests based in the time as it passes inside the framework.

5.6 Actor reasoning engine

The actor reasoning engine is used to make decisions regarding how to interact with the world the actor is situated in. The actor reasoning engine has a single iteration function that serves as the basis for its operation. This function can will evaluate the current world state of the actor and make recommendations depending on the results. It is expected that this evaluation will grow exponentially in computational intensity as additional world information is added to the world model used by the actors. Therefore, the reasoning engine will only make a new recommendation if the world has changed sufficiently from the previous state. The way this is done is by storing the sum of desire values in the actor state and calculating how much this deviates from the last time a selection was made. This deviation is called the threshold value, and is set by the designer to indicate how big the change in actor state must be to warrant a new action recommendation to the agents subscribing to the actors notifications.

If it is decided that the actor state has not changed sufficiently, the actor is referred to as being in a *waiting state.* This simply means that for this iteration, the reasoning engine will do nothing, and will most likely continue to do nothing if the world does not change in any meaningful way. If the reasoning engine decides that the world has change sufficiently from the previous iterations, it will perform a series of steps to make a recommendation. First, a *desire group* is selected; which is described further in section 5.6.1. Once a *desire group* is chosen, the reasoning engine attempts to identify the smart object that can provide the highest possible utility at this moment, this is explained in detail in 5.6.2. Once a service is found, it could be contested by other actors. If so, a conflict will be initiated against one of these actors. Should a smart object not be contested, a notification is sent to all agents subscribing to activation requests, informing them of the recommendation from the reasoning engine. Section 5.6.3 describes how a conflict is initiated. Figure 15 provides a flowchart describing the operations described in the previous paragraphs.

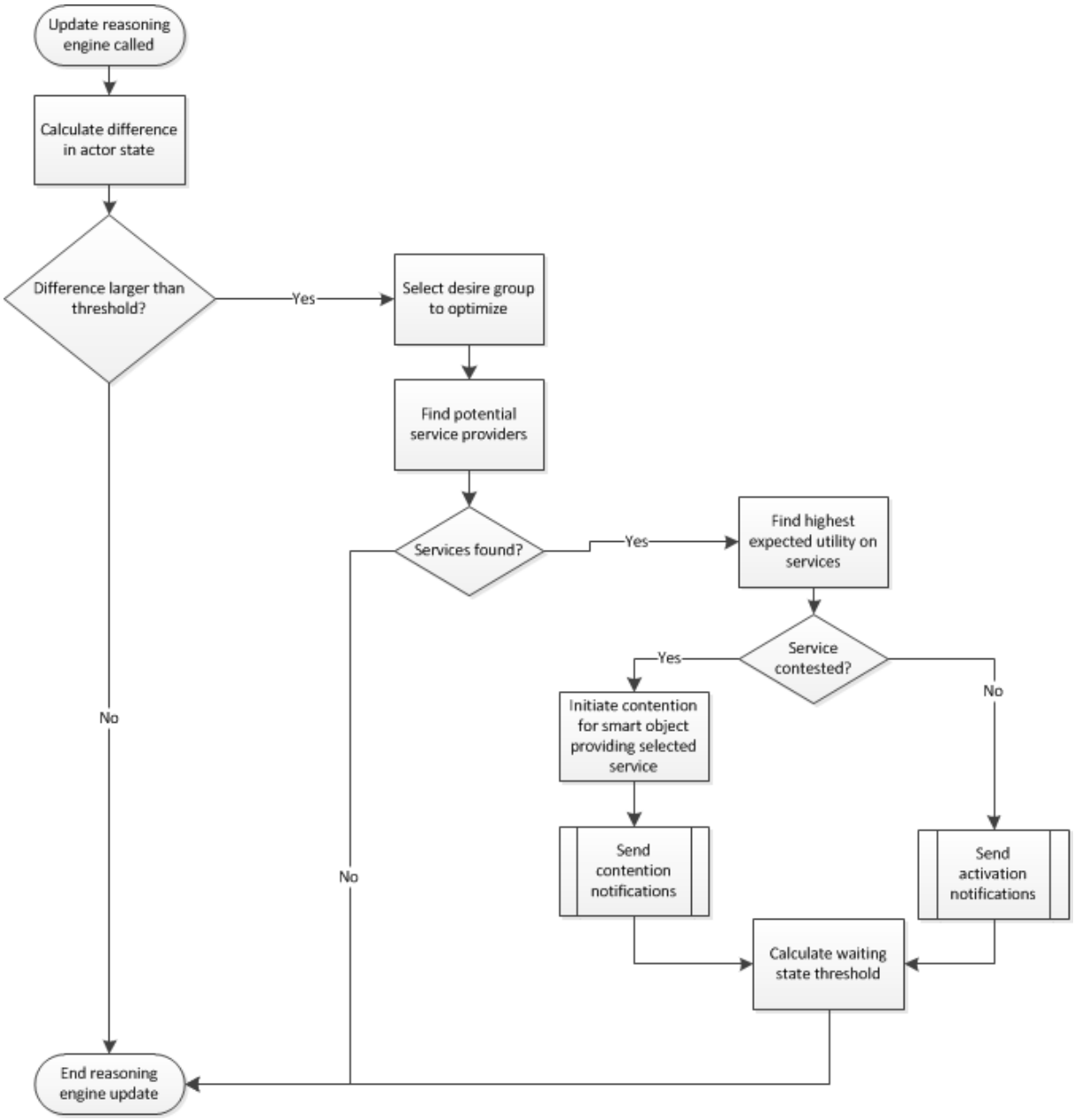


Figure 15- Single reasoning engine iteration

5.6.1 Selecting desire group to optimize

The reasoning process for choosing what desire group to optimize draws inspiration from neural network activation values. Each activation group can be thought of as a single neuron, where each attribute's activation value is a weighted input value. The reasoning process computes the activation level of each desire grouping (wealth, power and prestige). The grouping that has the highest rated activation function will be chosen for improvement. The rating for each desire type is calculated as follows:

Given $\boldsymbol{n}$ *contributing attributes where* $\boldsymbol{n}$ *is the number of attributes in the given desire grouping. The activation rating for a given desire type is the sum of all* $\boldsymbol{n}$ *attributes where the activation value generated by each individual trait is calculated by the remaining value of a given satisfaction function* $\boldsymbol{f}$ *for each attribute, multiplied by a given weighting function* $\boldsymbol{\omega}$*. The total activation rating is then normalized by* $\boldsymbol{n}$ *to account for the variation in attributes for any given desire group.*

$$\frac{1}{n}\sum_{i=0..n}^{n}(1-f_i)\cdot\boldsymbol{\omega}_i$$

Note the differences between neurons and this activation function. Firstly, the value of each input function (assumed to be between 0.0 and 1.0) is used as a negative value in this implementation. This means that the higher the satisfaction level returned from the function that produces the input value, the less chance it is for this desire group to activate. Second, to reduce the impact the amount of input values has on the overall activation value, the activation value is divided by the number of input values. This will cause desire groups with few but heavily weighted attributes to have a higher chance of being activated than an attribute set of many weakly weighted attributes. Once a desire type has been chosen, the actor queries the world model for smart objects offering services that can improve the attributes that affect the given desire.

5.6.2 Finding highest expected utility

Once the list of objects supplying services that affect the desire grouping has been identified, objects are tested to see if they can in fact be used by the actor by comparing attribute values as described earlier. The final step of selecting a service provider is to calculate the expected utility for each provider and selecting the highest value.

To find the highest expected utility, each smart object offering a service $\boldsymbol{s}$ *has its potential gain* $\boldsymbol{\phi}$*, multiplied with the actor's weight function* $\boldsymbol{\omega}$*. The expected utility is then computed by multiplying the potential gain by the product of the* $\boldsymbol{n}$ *contestants' relation value* $\boldsymbol{\psi}$*.*

$$\phi_s.\omega_s\prod_{i=0..n}^{n}\boldsymbol{\psi}_n$$

Once the process of selecting the service to utilize has been completed successfully, the reasoning engine will fire off an event to all listening agents, informing what service will be accessed. The selection process can fail in two ways: If there are no services that match the

selection criteria decided upon by the desire selection, or if the services available are contested by other actors. In the case of the former, the reasoning engine will simply stop running for the time being, doing nothing for this iteration of the selection process. It will behave as if in a *waiting state* until update is called again. In the case of the latter, the reasoning engine will initiate contention for the smart object offering the service.

5.6.3 Initiating contention

Contention is initiated when one or more actors, that are not a part of the reasoning engine's actor hierarchy, are listed as actively making use of, or attempting to make use of the same smart object (any service of the object will do) as the actor has itself selected for use. The reasoning engine will then use the same algorithm as used to select highest utility service, to select the best possible *conflict resolution strategy*, based on relation and attributes stored as part of the *conflict behavior*. Once a behavior has been selected, the world model will be queried to see if any other actors in the hierarchy of the reasoning engine are engaged with the same *conflict resolution strategy,* on the same smart object. If this is the case, the reasoning engine will send a notification to all subscribing agents that it recommends joining the conflict. If no agents in the same hierarchy are involved in conflict of the object, a new *conflict behavior* is created, and notifications are sent to all subscribing agents that a new conflict has been initiated. Once this has been done, the new change threshold is calculated and the reasoning engine enters *waiting state*.

5.6.4 Waiting state and action requests.

Unless the reasoning engine was unable to make a choice at all, some form of event will have been sent to all listening agents, informing them of a possible way to change the virtual world in a fashion that suits the actor they are tied to. If so, the reasoning engine will store the current *total utility* and enter a *waiting state*.

When the reasoning engine enters the *waiting state,* it will initialize a threshold variable that will slowly be reduced as time passes. This threshold defines the total change required to take place in the desire values of the agent before a new plan is formulated. As time passes, the total change required for a new plan to be formulated approaches zero. The actual desire values of an actor is calculated as often as possible, however this task automatically yields to higher priority tasks in the system. Also, actors higher up in the hierarchy are calculated first, leading to faction level decisions always being able to respond to changes in the world, while

actors with less importance calculate their desires more seldom if computational resources are tight. In addition to responding to the threshold being exceeded, the reasoning engine may also respond to having an action applied to it. This occurs at any point where the designer activates an action on an entity which has been added to the actor event list, not to be confused with the notifications described previously in this section. When an action is applied to an actor, the desire values of the actor is recalculated immediately and checked against the current plan threshold.

An actor may be given an action request from a parent actor. If so, both the parent and the actor which has been given the request store the expected action in their reasoning engine. As with other relationships, the actor is only expected to do the best possible job to fulfill the request. Because of this, the result of each request is undefined by nature. There is no way to demand that a request is fulfilled with absolute certainty. This once again ties in with the concept of solidarity. While this implementation allows independence in actors to some degree, it does not allow for actors to reevaluate their relationships with other actors as suggested by Medler et al. [22].

5.7 Summary

This chapter has provided a detailed description of the FIF design. The entirety of the framework architecture was shown in Figure 13, which in turn was described in detail throughout the chapter. It was shown how one would tie together the FIF and the virtual world it is to inform, through use of composition, as well as how these enhanced game objects would benefit the game through additional world data for their agents to utilize.

The internal world model for the FIF was described, which further informs the actor reasoning engine. The actor reasoning engine at the heart of the FIF is then allowed to evaluate the world as understood by the FIF, and provide recommendations in the form of service requests of behavior mobilizations. This process, which lies at the heart of the FIF, was shown in Figure 15. The behavior mobilizations and service requests are the final product of the FIF, as they can be easily translated into gameplay scenarios in a virtual world.

The FIF provides dynamic gameplay generation to game worlds through the enhanced world detail and simulation of factions. While game developers are still required to build the rules and representations of their virtual world, the FIF offers a way to create scenarios for these

rules and representations to be utilized, without having to script each scenario by hand. Actors with the same interests that are not allied, will automatically escalate and deescalate the tensions between each other and utilize different behavior strategies to attempt to get ahead in the virtual world. By tying game mechanics to behavior strategies, game developers will get autonomously generated conflict scenarios. These gameplay scenarios will also have the benefit of being different every time, given that the player or players are able to manipulate the attributes desired by actors in the world. Chapter 6 will describe how the design provided in this chapter was used to implement a prototype version of the FIF in C#.

Chapter 6 Framework implementation

In this chapter, we describe a proof of concept implementation of the FIF, which was developed using Visual studio and C#. C# was chosen for its native implementation of delegates and events, making it simple to create the notifications described in Chapter 5. This chapter will describe how the features of the design were implemented in practice, and provide a brief overview of the architecture.

Section 6.1 provides a short description of C# features that were essential to the implementation. Next is an overview of the different features, provided with class diagrams generated by visual studio from the complete solution. This provides an exact overview of the implementation. Section 6.2 describes the implementation of semantics and smart objects. Together with section 6.3, these two sections describe the functionality used to describe the virtual world to the FIF as described in section 5.1.

Section 6.4 describes how actors were implemented in the prototype, while section 6.5 shows how actor relations are defined. Next, section 6.6 describes in detail, how one uses the prototype to describe a virtual world to the FIF. Finally, a discussion of the result is provided in 6.7.

6.1 C# features

C# is a high level language developed and maintained by Microsoft. Thanks to the Mono project, the language has become one of the most widely used and platform independent solutions available [68]. In addition to this, C# is also one of the scripting languages usable directly in Unity3D, making it trivial to merge FIF with the test scenario. It should be noted that the C# features described in the following subsections are not exclusive to the language. C++ and other languages also provide extensions or native support for these.

6.1.1 Delegates and events

Delegates in C# provide the same functionality as function pointers in C#, but they hide the inherent complexity that rise when trying to use function pointers with member functions [69, 70]. Delegates are defined in the same way as any data structure. Figure 16 shows how a delegate is declared in visual studio 2010.

```
public delegate void ServiceActivationRequest(
    object sender, ServiceActivationRequestArgs args);
```

Figure 16 - Delegate declaration in visual studio 2010

Events are in essence syntactic sugar for delegates. By declaring an event instance of a delegate type, one essentially gets a list of function pointers that can be subscribed to by any function that has the same signature as the delegate. Should a game developer wish to subscribe to an FIF event, she has simply to create a function with the signature described by the delegate. Once created, the signature can be added to the event instance, and invoked by the framework as can be seen in Figure 17.

```
ServiceActivationRequestArgs args = new ServiceActivationRequestArgs
                                    {
                                        InterestedActor = actor,
                                        Service = serviceToUse,
                                        Owner = so
                                    };
if (OnServiceActivationRequested != null)
{
    OnServiceActivationRequested(this, args);
}
```

Figure 17 - Invoking a service activation event

6.1.2 Internal keyword

The internal keyword is an accessibility keyword specific to the C# language. This keyword attempts to replace the C++ keyword “friend” which allows one class to access protected variables from the class that has friended it. The internal keyword is less restrictive, allowing all components of the same assembly to benefit from this extended privilege [71]. While this keyword breaks with strict object oriented principles, it allows for special behavior patterns to be enforced. For instance, by making the semantic attribute class internal and making the constructor protected, one can make sure that only the accessor system (explained in section 5.1.2) is able to attach attributes to objects.

Thanks to the internal keyword, it is simple to enforce policies that make it less likely that the framework will be used incorrectly. In addition, it allows for designs where one can do separation of concerns without having to duplicate the data set being worked on, which is useful when attempting to keep the memory footprint of the application at a minimum.

6.1.3 Properties

A property is syntactical sugar for implementing get and set functions in C#. Both advanced get and set functions, with bodies supporting all types of programming logic, as well as automatically implemented get and set functions are supported [72].

6.2 Implementing semantic attributes and smart objects

Smart objects and *semantic attributes* are closely interconnected in the core design described in Chapter 5. Therefore, these two features of the framework implementation are described as one feature set. Figure 19 describes in detail how *smart objects*, *services* and *semantic attributes* are implemented, as well as the relationship between them. Note that this class diagram is generated by visual studio 2010 directly from the code. Because of this, some descriptors are not UML compliant; hash symbols marks the entry as internal, as described in section 6.1.2. Where a list or other container is followed by the new keyword and a constructor, the container uses automatic initialization.

Figure 19 shows the containers for services, attributes and interested actors are all marked as internal. This allows the rest of the faction interaction system to directly manipulate these data structures should it be necessary. For instance, when the *actor reasoning engine* is searching for a smart object that is capable of optimizing its selected attribute, it will access both the services list to evaluate the potential return, as well as the interested actor list to calculate the potential contestants for the smart object in question.

Semantic attributes are very simple constructs, as can be seen from Figure 19. They contain a name, the current internal value, as well as a defuzzifier. The defuzzifier is where the designer can make modifications to create any kind of *membership function* as described in section 3.6. The defuzzifier interface is the only requirements for how these functions are implemented, allowing the designer to take any game state into account when computing the defuzzified value. An example of this is provided in section 7.3.

Thanks to the internal keyword, and protected constructors, the only way to attach an attribute to a smart object is through the attribute accessor class, as can be seen in Figure 18. This is the tool used by game designers and developers when attempting to describe smart objects. The attributeMap in the accessor contains a hashmap of lists. The hashmap uses attribute names as an index, which in turn returns a list of all objects in the world that contains the attribute. This

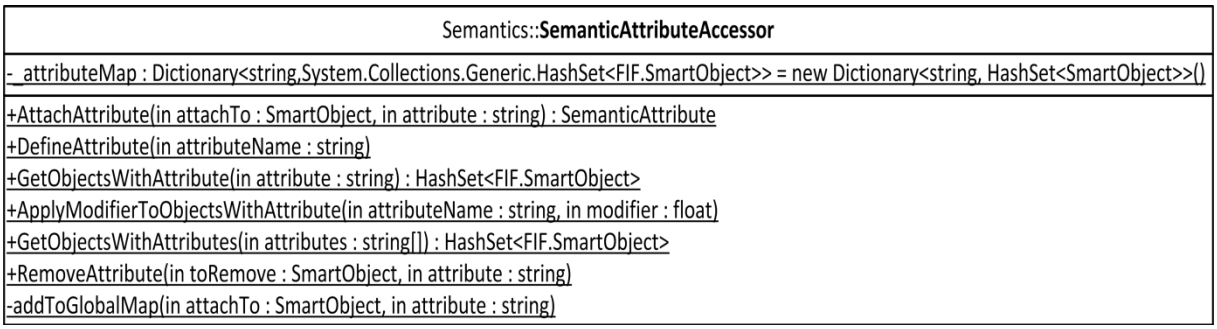


Figure 18 - Semantic attribute accesor implementation

allows both the FIF and the game to quickly leverage the semantic information to find objects of interest.

6.3 Implementing the world representation

The world model implementation was combined with the behavior model to contain all information that was required to describe the world. The different containers seen in Figure 20 contain all semantic attributes that are used to describe conflict in the system (added by designers). The "behaviors" container contains all possible conflict resolution strategies that can be applied. Note that these behaviors are not instances active, but the abstract representation referenced by all instances of the conflict.

Functionality was also added to assist in selecting the best possible behavior strategy, through the "GetBestFitMobilization" function, which returns either a new instance or one already being applied by other actors in the actor hierarchy. When starting to use the FIF implementation, one first sets a new world model in the "FactionInteractionSystem". Figure 21 shows how this class is implemented. As can be seen, it implements the singleton pattern, and contains an instance of the world model, a list of all actors in the scene and all active mobilizations. The "FactionInteractionSystem" is responsible for running the entirety of the implementation through its update function. When this function is called, it will iterate through all actors in the world and allow them to perform reasoning.

6.4 Actors

In Figure 22 one can see an overview of the actor and actor reasoning engine implementation. These are split in two parts, where a reasoning engine is informed of the actor state through its update function. It should be noted that this implementation does not include the ability to propagate requests down the actor hierarchy, as described in section 5.1.3. This feature is closely tied to the *solidarity* functionality discussed in section 5.6. This functionality can still be implemented by one making use of the framework, should she wish to do so. This can be accomplished by adding event listeners to the "commander" actors and having these relay requests to agents subscribing to actors further down the hierarchy.

Figure 19 - Smart object implementation

FIF::**WorldModel**
-_factions : Dictionary<string,int> = new Dictionary<string, int>() -_wealthAttributes : List<FIF.Semantics.SemanticAttribute> = new List<SemanticAttribute>() -_prestigeAttributes : List<FIF.Semantics.SemanticAttribute> = new List<SemanticAttribute>() -_powerAttributes : List<FIF.Semantics.SemanticAttribute> = new List<SemanticAttribute>() -_behaviors : List<FIF.BehaviorModel.Behavior> = new List<Behavior>()
+WorldModel() +DefineWealthAttribute(in attributeName : string) : SemanticAttribute +DefinePowerAttribute(in attributeName : string) : SemanticAttribute +DefinePrestigeAttribute(in attributeName : string) : SemanticAttribute -defineAttribute(in attributeName : string, in list : List<FIF.Semantics.SemanticAttribute>) : SemanticAttribute +CreateNewWealthInstance(in attributeToGet : string) : SemanticAttribute +CreateNewPrestigeInstance(in attributeToGet : string) : SemanticAttribute +CreateNewPowerInstance(in attributeToGet : string) : SemanticAttribute -createNewAttributeInstace(in list : List<FIF.Semantics.SemanticAttribute>, in attributeToGet : string) : SemanticAttribute +GetPrestigeAttributes() : List<FIF.Semantics.SemanticAttribute> +GetPowerAttributes() : List<FIF.Semantics.SemanticAttribute> +GetWealthAttributes() : List<FIF.Semantics.SemanticAttribute> +AddBehavior(in behavior : Behavior) +GetRelation(in mainActor : string, in toRelate : string) : float +GetBestFitMobilization(in actor : Actor, in contestant : Actor) : Behavior -actorHasRequiredAttributesForBehavior(in actor : Actor, in behavior : Behavior) : bool

Figure 20 - World model

FIF::**FactionInteractionSystem**
-_instance : FactionInteractionSystem -_world : WorldModel -_actors : List<FIF.Actor> = new List<Actor>() -_mobilizations : List<FIF.BehaviorModel.BehaviorMobilization> = new List<BehaviorMobilization>()
+World() : WorldModel +GetInstance() : FactionInteractionSystem #FactionInteractionSystem() +ApplyGlobalModifierOnActors(in attribute : string, in modifier : float) +ApplyGlobalModifierOnSmartObjects(in attribute : string, in modifier : float) +Update() +SetWorldModel(in worldModel : WorldModel) +AddActor(in actor : Actor) +GetMoblizationsOn(in target : SmartObject) : List<FIF.BehaviorModel.BehaviorMobilization> +GetInterferingMobilizations(in target : SmartObject, in interestedActor : Actor) : List<FIF.BehaviorModel.BehaviorMobilization> +CanContest(in target : SmartObject, in interestedActor : Actor) : bool -conflictBehaviorListener(in sender : object, in args : BehaviorActivationRequestArgs) +RemoveActor(in actor : Actor) +CeaseContentionFor(in contestingObject : SmartObject, in instigator : Actor) +EndParticipationIn(in mobilization : BehaviorMobilization, in participant : Actor) +SelectNewOwnerFor(in mobilization : BehaviorMobilization) : bool

Figure 21 - Faction interaction system

There are several features of note in Figure 22. There are many functions that allow a designer to directly modify the attribute values in the actor state, such as the *"GetAttribute"* and "*ModifyAttribute"* functions. Designers can also directly modify relationship values and get

the current desire values. The actor reasoning engine requires advanced functionality for querying the state of other actors. For instance, the reasoning engine must be able to find the relation of all other actors interested in a smart object when attempting to decide what object to recommend accessing. This is accomplished by storing a static list of all actors inside the actor itself. The actor reasoning engine is a complex system that changes behavior by minute variations to the world around it. The update function takes into account large amounts of the world information, as described in section 5.6.2. This implementation has potential for causing performance issues, which is discussed in section 7.4.

6.5 Faction relations

Relationships between actors are rather problematic to set up when the amount of actors increases quickly. Assuming that one would have to set up relationship values for all actors in the scene the amount of values to set would be N*(N-1). This would quickly become problematic for designers to cope with, so a different solution was required. For this implementation, two measures were taken to reduce the required work.

First, a data structure to hold a full relation table was created, allowing for the same relation table to be copied between several actors that should hold the same relations. Second, the actor will send a request up its hierarchy (if it has one), querying any actors further up if they have a relationship value to the actor. Should a relationship value be found further up the hierarchy, this value will be adapted; if not the standard initialization value of 0.5 (neutral) will be used. This way, actors can be added to the game while it is running, with the actors being related to, depending on their ties to other actors.

FIF::**Actor**
-_relations : Dictionary<string,FIF.ActorRelation> = new Dictionary<string, ActorRelation>() -_world : WorldModel -_reasoner : ReasoningEngine #_state : ActorState -actorCounter : uint = 0 -_actors : Dictionary<string,FIF.Actor> = new Dictionary<string, Actor>() -_actorIndex : uint
+ActorUniqueName() : string +OnServiceActivationRequested() : ServiceActivationRequest +OnConflictBehaviorInitiated() : BehaviorActivationRequest +Wealth() : float +Prestiege() : float +Power() : float +Parent() : Actor +Root() : Actor +Actor(in world : WorldModel) +Actor(in world : WorldModel, in attributeSet : ActorAttributeSet, in actorUnqiueName : string) +GetActor(in actorUniqueName : string) : Actor +GetRelation(in actor : Actor) : float -createNewRelation(in actor : Actor, out relation : ActorRelation) +Update() +AttachWealthAttribute(in attributeName : string) : SemanticAttribute +AttachPowerAttribute(in attributeName : string) : SemanticAttribute +AttachPrestigeAttribute(in attributeName : string) : SemanticAttribute +ModifyAttribute(in attribute : string, in modifier : float) +HasAttribute(in attribute : string) : bool +CalculateTotalDifference(in potentialBehavior : Behavior) : float +SetRelation(in actor : Actor, in relation : float) +ResetRelation(in newRelationSet : Dictionary<string,FIF.ActorRelation>) +GetAttribute(in attributeName : string) : SemanticAttribute +GetPowerAttribute(in attributeName : string) : SemanticAttribute +GetPrestigeAttribute(in attributeName : string) : SemanticAttribute +GetWealthAttribute(in attributeName : string) : SemanticAttribute -getSemanticAttribute(in attributeName : string, in attributes : List<FIF.Semantics.SemanticAttribute>) : SemanticAttribute

ActorReasoning::**ReasoningEngine**
-_world : WorldModel -_fis : FactionInteractionSystem -_maxWaitingStateInertia : float = 0.25f -_inertiaRetentionPerUpdate : float = 0.995f -_currentWaitingStateIntertia : float -_currentAttributeList : List<FIF.Semantics.SemanticAttribute> -_waitingStateValue : float
+OnServiceActivationRequested() : ServiceActivationRequest +OnConflictBehaviorInitiated() : BehaviorActivationRequest +ReasoningEngine(in world : WorldModel) +Update(in actor : Actor) -resetWaitingState(in newWaitingStateValue : float) -otherFactionsWillContestObject(in so : SmartObject, in actor : Actor) : bool -initiateContention(in actor : Actor, in so : SmartObject) -createNewBehaviorMobilization(in actor : Actor, in so : SmartObject, in contestant : Actor) : BehaviorMobilization -selectSmartObject(in actor : Actor, out serviceToUse : Service) : SmartObject -calculateConflictMultiplier(in actor : Actor, in candidate : SmartObject) : float -getAttributesToActivate(in state : ActorState, in wealth : float, in power : float, in prestige : float) : List<FIF.Semantics.SemanticAttribute>

Figure 22 - Actor implementation overview

6.6 Defining a world

This prototype implementation of the FIF allows the user to define all the features described in Chapter 5. This section describes how the prototype implementation can be initialized to perform the operations discussed in the design.

1. Create a new world instance and set it in the faction interaction system definition.
2. Define the attributes relevant to the world representation and register them with world instance.
3. Define the behaviors applicable to the world and add them to the faction interaction system.
4. Create all actors that will participate in the world and register them in hierarchies as appropriate.
5. Define the faction relations between all actors in the scene.
6. Add actors to the world.
7. FIF is now ready for use.

Once these steps have been accomplished, the FIF can be updated in its entirety by simply calling the update function on the faction interaction system. This will cause all actors to be updated and post recommendations to the game system.

The recommendations generated by the FIF are the final product of the system. These are either service activation requests or behavior mobilization requests, sent to any agents that subscribe to the actors' events. It is through these requests that gameplay is dynamically generated, as agents respond (or ignore, depending on the implementation made by the game developer) to the requests. As requests are fulfilled, the attributes requested change, and new requests are posted, resulting in an endless succession of scenarios emerging from the system.

6.7 Summary

The faction interaction framework prototype was implemented as a standalone library of functionality that can be leveraged by any game capable of utilizing .NET/CIL libraries. By defining a world with FIF, and tying agents and smart objects to the framework, the FIF is capable of adding an additional layer of interaction with the world it is implemented in.

Chapter 7 utilizes the work presented in this chapter to build a small virtual world for testing the feasibility of the prototype. Here, the hunger example presented in section 1.6 is implemented on a small scale, to create a world where two villages are fighting for access to the only food source available.

Chapter 7 Game scenario implementation

The prototype described in chapter 6 fulfills the requirements presented in Chapter 5. In this chapter, we evaluate FIF using a virtual world based on the *"hunger conflict"* example described in section 1.6. This chapter describes the different features of the example and how they are implemented using a combination of functionality provided by Unity3D and the faction interaction framework. Figure 23 shows the game scene from the perspective of the green faction. To the right, marked by the widget, is a green warrior agent that has decided to optimize the battle ready attribute. Off in the horizon can be seen the main food source representation, as well as a large group of faction members in conflict concerning the object.

Figure 23 - Unity3D scene view of the world

There are several key features of the *"hunger conflict"* example. These are described in the following way; Section 7.1 describes the implementation of food sources, the main conflict area of the game scenario. Next, section 7.2 covers how the villages are set up and defined for FIF. Section 7.3 describes the implementation of actors and agents to create a world with active participants. Once the world has been created, the simulation is run for 2 minutes, in which the FIF can be observed to be generating behavior and service activation requests.

The functioning of the framework is discussed in section 7.4, which points to the service and behavior activation requests that are posted by the framework during a single run. Section 7.5

then deals with the performance side of the framework, looking into whether the FIF can be said to be applicable to real time systems like games and simulations. Finally, section 7.6 briefly discusses the prototype in regards to the problem statement presented in section 1.4, before a chapter summary is provided in section 7.7.

7.1 Food sources

Implementing food sources in the game scenario was done by creating a geometric primitive using the unity world editor. Once the primitive was placed, a smart object (from the FIF implementation) and an AI module was attached to it. When the game starts, the AI module uses attribute and service accessors to add a service that provides hunger satisfaction, making it known that this entity can satisfy the particular need.

By attaching an activation function in the AI module to the smart object, the food source became capable of reacting to actors attempting to activate it. Figure 24 describes the structure of the AI component. This component is part of the game logic as is commonly used in Unity3D, while also interacting with the FIF. Should an agent activate the food source, the AI component would apply the action result of the service being activated, resulting in the hunger satisfaction value of the actor, connected to the agent, to be increased.

UnityGame::**FoodSourceAiComponent**
+HungerSatesfactionProvided : float = 0.25f +DataRefreshTime : float = 1f/15f +TimeBeforeResolvedConflictIsDeleted : float = 15f -_resolvedConflicts : List<ResolvedConflict> = new List<ResolvedConflict>() -_dataRefreshTimer : float
+FoodSource() : SmartObject +Activated() : bool -Start() -worldActionActivatedEvent(in sender : object, in eventArgs : WorldActionEventArgs) -Update() -refreshMobilizationData() -addResolvedConflict(in resolvedConflict : BehaviorMobilization) -UpdateMobilizations()

Figure 24 - Food source ai component

Two variations of the game scenario were made. The version displayed in Figure 23 contains a single, major, food source, as provided by food drops. The second version of the solution

also had two food sources representing fields belonging to each village. The difference between the two implementations will be discussed in the summary of this chapter.

7.2 Villages

Villages were created by combining small sets of primitives into houses, and then creating a small collection of these new entities. The entirety of the village had a smart object attached to it with a faction attribute, describing what faction it belonged to. This allowed agents to easily find their way home. In addition, this allowed for a complex "bravery" attribute to be created. The bravery defuzzifer created for this attribute takes into account the surroundings of each agent currently subscribing to the actor. The more enemies in the area, the lower the attribute would become, finally causing the actor to request the optimization of bravery, causing the agents to run home to their village.

One of the village structures had an additional smart object attached to it; listing the service "provide battle readiness" this service was only interesting to warriors of the group in question. This service was added mainly to increase the detail of the world, as well as allowing changes in priorities to be observed as warriors grew hungry.

7.3 Actors and agents

The game scenario was implemented with only two faction hierarchies. Each actor in the hierarchy was assigned one or more agents that subscribed to their request events. In each of these hierarchies, the following actors are implemented:

- Faction parent actor
- Elders
- Warriors
- Thieves

Agents were implemented in the game world using physics primitives, rendering primitives and the path finding library of Unity3D. To bring the agents to life, a small, general purpose FSM was built to handle conflict resolution and service utilization requests created by the actors. This FSM contained generalized behavior for FIF events, as well as specialized states for handling bravery and idleness (no recommendation from the actor). Figure 25 describes

how the FSM functions in each agent. Implementing this functionality was fairly simple, thanks to the fact that all reasoning on when to activate each state was handled by the faction interaction framework through actors posting requests to the agent FSMs.

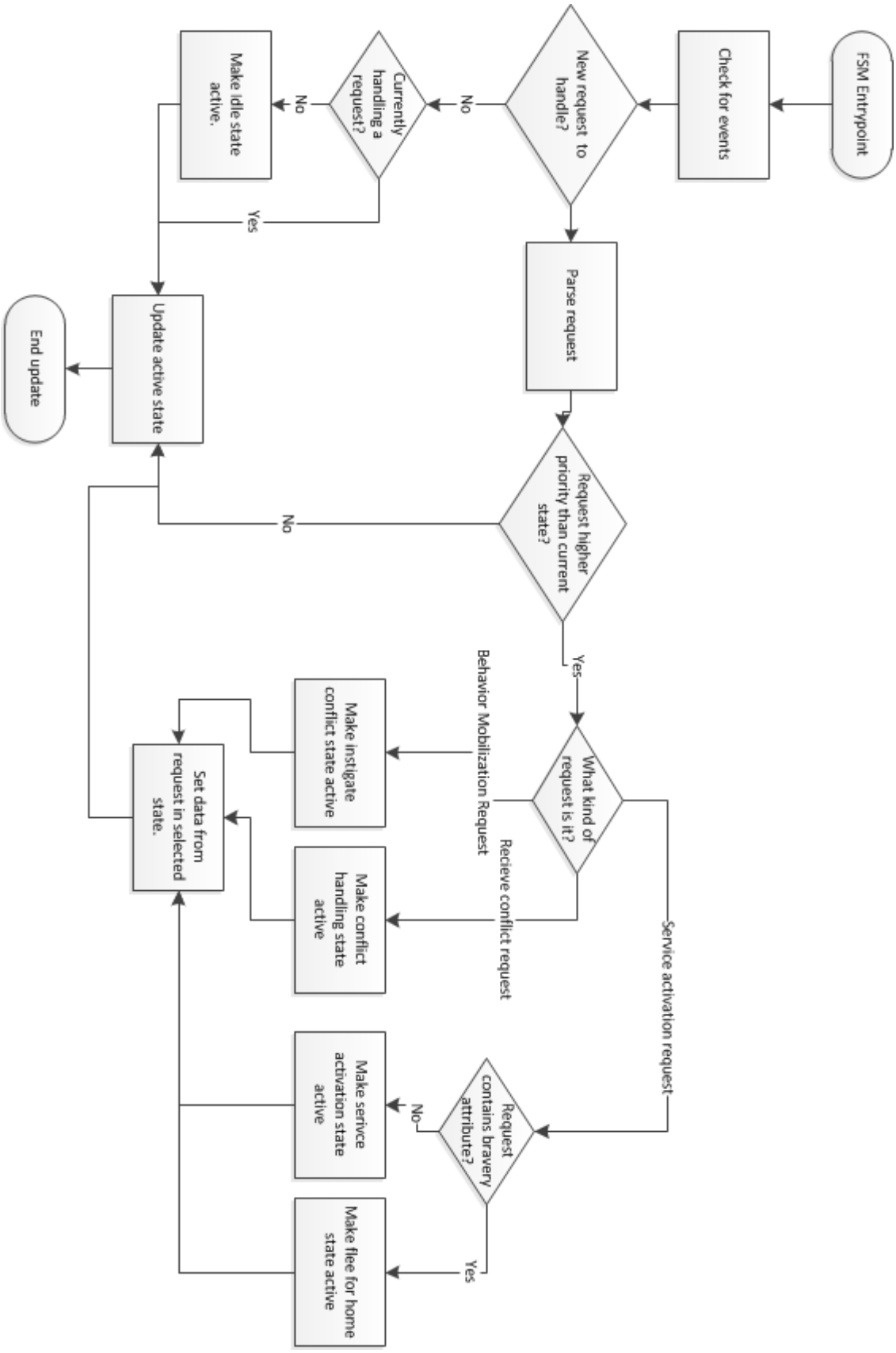


Figure 25 - FSM overview

7.4 Observing faction interaction

While possible to simply observe the agents as they interacted with the world in accordance to the recommendations of the FIF, it was easier to see the effects of actors on the world through tracking change in FSM state and events sent to them. By using the built in unity GUI library, a simple overlay was created to report all events posted by actors, and all FSM changes made by agents.

Figure 26 shows the game scenario approximately 60 seconds after the simulation was started. The left overlay shows the current desire values for all actors in the scenario. Note how prestige values are set to 0. The reason for this value is that the current scenario does not have any desire attributes for prestige. The result of this is simply that the actor will never attempt to optimize prestige.

The right hand side of the figure shows the last recommendation made by the different actors in the scene. Each time an actor makes a new recommendation, this overlay is updated. As can be seen from the overlay, the different factions are opting out of conflict in favor of slightly less optimal choices, in this particular configuration of the virtual world. Figure 27 shows the same game scenario. In this set, one can see the FSM view, which describes the current state active in every agent's state machine. Here one can see the different agents going about their activities, some attempting to access the middle food source, while the other faction racing to enter conflict with them over the precious resource.

Finally, in Figure 28, more time has passed, and the priorities of the actors have changed. As the degree of hunger has been set to increase over time in this simulation, the actors without access to the main food supply will become more and more interested in this resource. Looking at the center of Figure 28, one can see a sole green villager, being engaged in a heated argument by the warriors of the red village. While the other red villagers were content with waiting for the green villagers to get their food, the red warriors chose to engage the nearest green villager in conflict, due to their increased aggressiveness. This shows that small changes to the simulation will create completely different gameplay scenarios. Players affecting the simulation through game mechanics would therefore make sufficient impact on the simulation state to drive the generation of new scenarios forward.

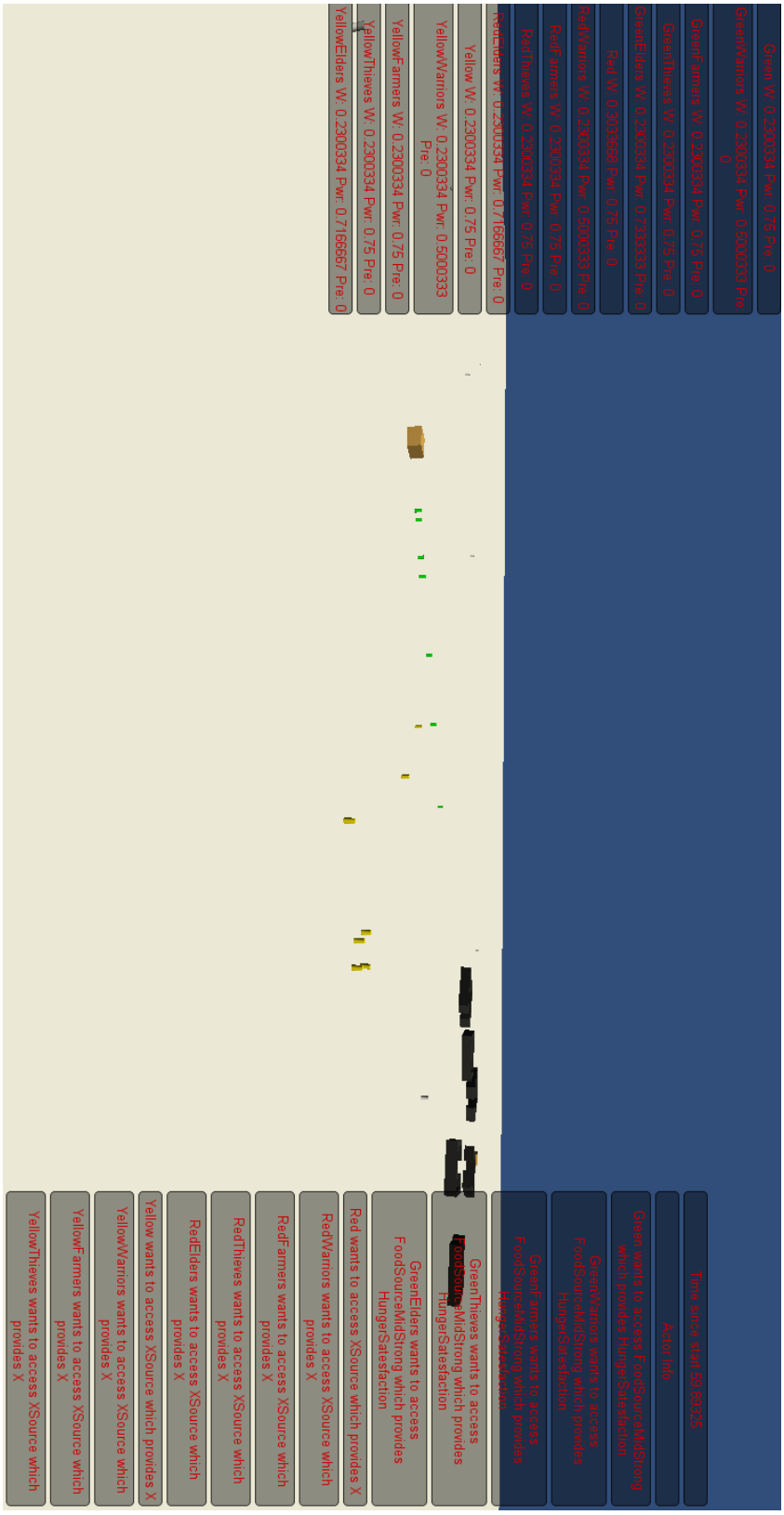


Figure 26 - Actor view in a single food source game scenario

Figure 27 - FSM view in a single food source game scenario

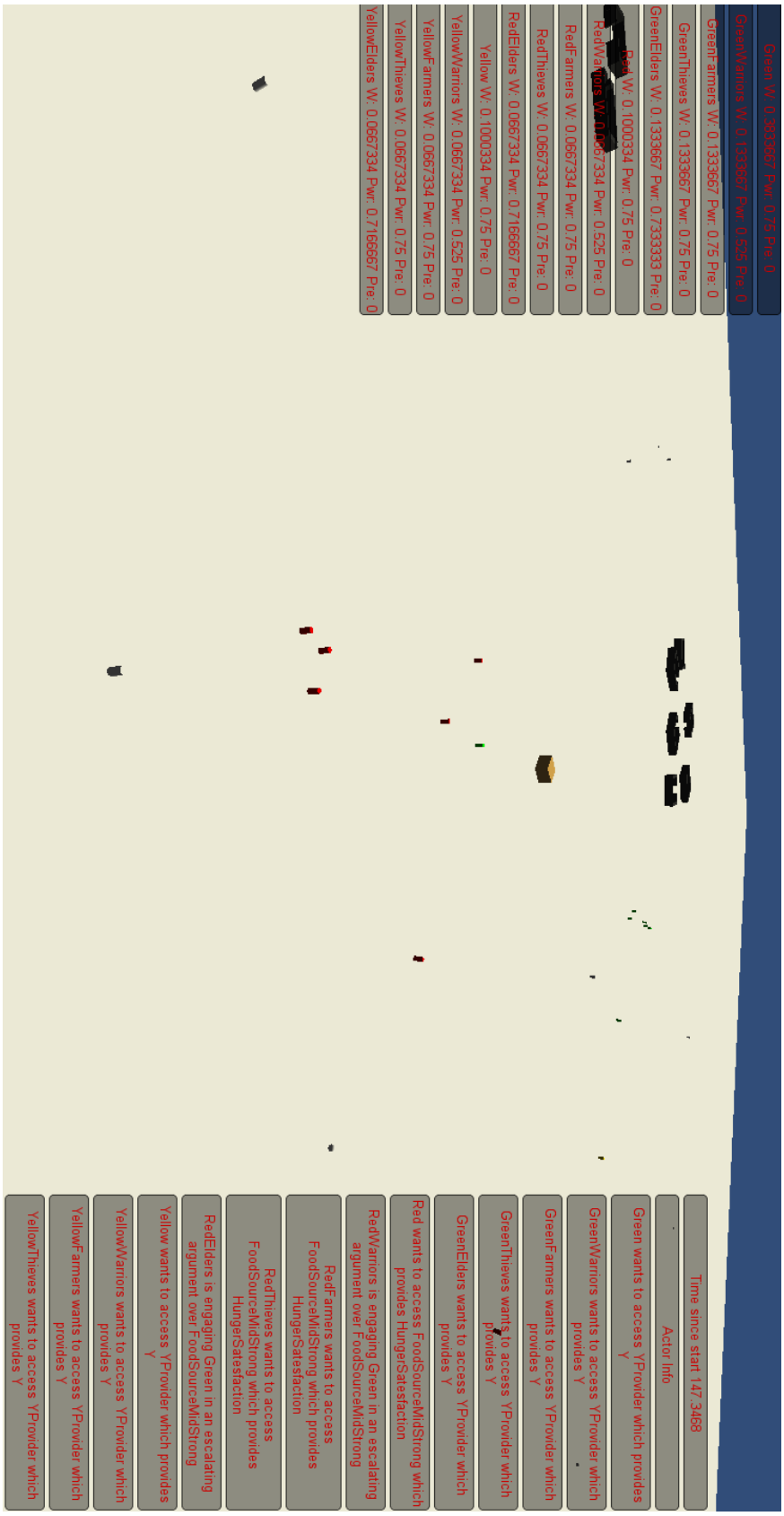


Figure 28 - Conflict between green and red

7.5 Investigating performance

In such a small scenario as the one provided, there were no hints of the implementation having any performance issues. Performance is measured using the internal unity3D profiler. The unity profiler uses several metrics to describe the performance of a solution [73], these can be seen in figure 27, and are as follows:

1. "Overview" lists function taking a measurable amount of time to compute in the setting. Note that the function "WaitForTargetFPS" is an idle function that will fill the remaining time of a frame to remain within the bounds of the locked frame rate of 60.

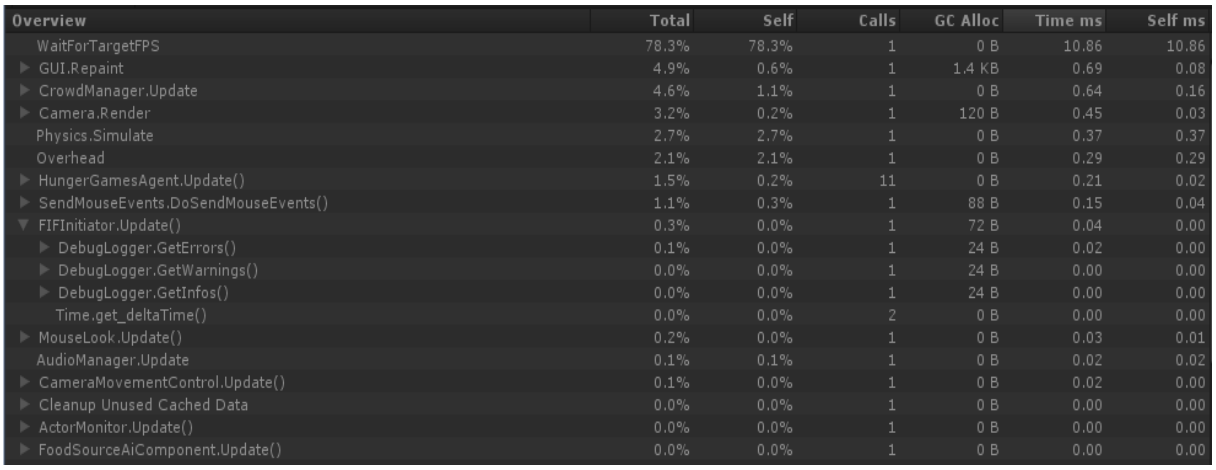
2. "Time ms" describes the update time taken to update the function, and all sub functions. As an example, observe the (FIFInitiator.Update) function in figure 27. Here one can see that the function itself takes too little time for the profiler to measure, however counting all sub functions, it takes a total of 0.04 ms.

3. "Self ms" is a measurement of how long the listed function took to execute, excluding any sub functions.

4. GCalloc describes the amount of memory allocated by the garbage collector for the given function. This metric is ignored for the purpose of this discussion.

5. The metrics "Total" and "Self" are the percentage representations of the total time taken for the given frame. In other words, they are the percentage versions of the "Time ms" and "Self ms" columns.

In Figure 29 one can see that the time spent updating the FIF (FIFInitiator.update) only takes 0.04 milliseconds (ms), where most of the time used is taken by the debug functionality. While this shows that there are no obvious flaws in the implementation, especially with such a small test scenario, this does not say much for the performance when attempting to scale the solution.



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
WaitForTargetFPS	78.3%	78.3%	1	0 B	10.86	10.86
GUI.Repaint	4.9%	0.6%	1	1.4 KB	0.69	0.08
CrowdManager.Update	4.6%	1.1%	1	0 B	0.64	0.16
Camera.Render	3.2%	0.2%	1	120 B	0.45	0.03
Physics.Simulate	2.7%	2.7%	1	0 B	0.37	0.37
Overhead	2.1%	2.1%	1	0 B	0.29	0.29
HungerGamesAgent.Update()	1.5%	0.2%	11	0 B	0.21	0.02
SendMouseEvents.DoSendMouseEvents()	1.1%	0.3%	1	88 B	0.15	0.04
FIFInitiator.Update()	0.3%	0.0%	1	72 B	0.04	0.00
DebugLogger.GetErrors()	0.1%	0.0%	1	24 B	0.02	0.00
DebugLogger.GetWarnings()	0.0%	0.0%	1	24 B	0.00	0.00
DebugLogger.GetInfos()	0.0%	0.0%	1	24 B	0.00	0.00
Time.get_deltaTime()	0.0%	0.0%	2	0 B	0.00	0.00
MouseLook.Update()	0.2%	0.0%	1	0 B	0.03	0.01
AudioManager.Update	0.1%	0.1%	1	0 B	0.02	0.02
CameraMovementControl.Update()	0.1%	0.0%	1	0 B	0.02	0.00
Cleanup Unused Cached Data	0.0%	0.0%	1	0 B	0.00	0.00
ActorMonitor.Update()	0.0%	0.0%	1	0 B	0.00	0.00
FoodSourceAiComponent.Update()	0.0%	0.0%	1	0 B	0.00	0.00

Figure 29 - Profile overview in single food source game scenario

Once the initial implementation had proven to run effortlessly, several attempts were made to increase load on the system. The initial attempt was to add a third village, with all the same features as the two other villages present in the game scenario. This would bring the total number of actors active in the scene to 15, and the number of agents to 21. This addition to the game scenario had miniscule effects in regards to performance, as can be seen in Figure 30. The time in milliseconds to update the FIF is listed as 0.03 ms, while the agent FSM logic

WaitForTargetFPS	76.2%	76.2%	1	0 B	10.57	10.57
CrowdManager.Update	5.5%	1.0%	1	0 B	0.77	0.15
Camera.Render	5.2%	0.1%	1	120 B	0.72	0.02
GUI.Repaint	3.7%	0.7%	1	1.4 KB	0.51	0.10
Physics.Simulate	2.8%	2.8%	1	0 B	0.38	0.38
HungerGamesAgent.Update()	2.6%	0.2%	21	314 B	0.37	0.03
Overhead	1.7%	1.7%	1	0 B	0.24	0.24
SendMouseEvents.DoSendMouseEvents()	1.1%	0.2%	1	108 B	0.16	0.04
FIFInitiator.Update()	0.2%	0.0%	1	72 B	0.03	0.00
DebugLogger.GetErrors()	0.1%	0.0%	1	24 B	0.01	0.00
DebugLogger.GetInfos()	0.0%	0.0%	1	24 B	0.00	0.00
DebugLogger.GetWarnings()	0.0%	0.0%	1	24 B	0.00	0.00
Time.get_deltaTime()	0.0%	0.0%	2	0 B	0.00	0.00
CameraMovementControl.Update()	0.1%	0.0%	1	0 B	0.02	0.00
MouseLook.Update()	0.1%	0.0%	1	0 B	0.02	0.00

Figure 30 - Performance when adding a third village to the game scenario

is taking 0.37 ms. These numbers are very small in comparison to the "CrowdManager.Update" (0.77 ms) which is responsible for path finding and collision avoidance for all the agents in the world. This makes sense when one considers that the reasoning engine mainly concerns itself with evaluating smart objects, as is explained in section 5.6. To quickly repeat the process of the actor reasoning engine, each actor must first select an attribute to optimize, and then find all potential smart objects that can supply the

given attribute, needing to consider the other actors in the scene, meaning that unless there are many objects to evaluate, the amount of actors will have little effect.

With this in mind, the next attempt of provoking performance issues added the following:

1. Two additional food sources added
2. Added attribute “X” to the game scenario, with five smart objects providing the service and a timed negative modifier. To increase conflict generation, the main food source is also made the strongest provider of this attribute.
3. Added attribute “Y” to the game scenario, with ten smart objects providing the service. No timed modifier was added to this attribute.
4. Disabled debug logging from the FIFInitiator system to get a better impression of release performance.

The result of these changes to the scenario was that the FIFInitator update dropped even further down the list in the profiler (due to debug writes being disabled). Figure 31 shows how the FIFInitiator update no longer has measurable computation time. While this scenario is still

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
CrowdManager.Update	30.7%	6.4%	1	0 B	0.88	0.18
GUI.Repaint	19.3%	2.7%	1	1.4 KB	0.55	0.08
Camera.Render	15.1%	1.0%	1	120 B	0.43	0.03
Physics.Simulate	14.3%	14.3%	1	0 B	0.41	0.41
Overhead	10.1%	10.1%	1	0 B	0.29	0.29
HungerGamesAgent.Update()	3.9%	1.1%	21	0 B	0.11	0.03
SendMouseEvents.DoSendMouseEvents()	1.7%	0.8%	1	68 B	0.05	0.02
Cleanup Unused Cached Data	1.1%	1.1%	1	0 B	0.03	0.03
MouseLook.Update()	0.8%	0.2%	1	0 B	0.02	0.00
CameraMovementControl.Update()	0.7%	0.2%	1	0 B	0.02	0.00
AudioManager.Update	0.6%	0.6%	1	0 B	0.01	0.01
YProviderAIComponent.Update()	0.3%	0.3%	10	0 B	0.01	0.01
FoodSourceAiComponent.Update()	0.1%	0.1%	3	0 B	0.00	0.00
List`1.get_Count()	0.0%	0.0%	3	0 B	0.00	0.00
Time.get_deltaTime()	0.0%	0.0%	3	0 B	0.00	0.00
Graphics.BeginFrame	0.1%	0.1%	1	0 B	0.00	0.00
Loading.UpdatePreloading	0.1%	0.0%	1	0 B	0.00	0.00
WaitForTargetFPS	0.1%	0.1%	1	0 B	0.00	0.00
ActorMonitor.Update()	0.0%	0.0%	1	0 B	0.00	0.00
ParticleSystem.Update	0.0%	0.0%	1	0 B	0.00	0.00
ProcessRemoteInput	0.0%	0.0%	1	0 B	0.00	0.00
XProvider.Update()	0.0%	0.0%	5	0 B	0.00	0.00
FIFInitiator.Update()	0.0%	0.0%	1	0 B	0.00	0.00
Time.get_deltaTime()	0.0%	0.0%	2	0 B	0.00	0.00
Network.Update	0.0%	0.0%	1	0 B	0.00	0.00
Physics.UpdateSkinnedCloth	0.0%	0.0%	2	0 B	0.00	0.00
BattlePreperationAiComponent.Update()	0.0%	0.0%	3	0 B	0.00	0.00
HandleUtility.SetViewInfo()	0.0%	0.0%	1	0 B	0.00	0.00
MeshSkinning.Update	0.0%	0.0%	1	0 B	0.00	0.00
Physics.Interpolation	0.0%	0.0%	2	0 B	0.00	0.00
Substance.Update	0.0%	0.0%	1	0 B	0.00	0.00

Figure 31 - Extended scenario performance overview

very small in comparison to open world games, it still provides insight into the performance of the solution. Figure 30 shows another screen capture from the Unity3D profiler, this one

showing a graph of the last 300 frames of a simulation. "*WaitForTargetFPS*" was highlighted in yellow to show the amount of time the game engine spent idle during the simulation. The chart is interpreted by looking at the highest point of the colored area. This section shows how many frames per second is currently possible, offset to always be 60 by the "*WaitForTargetFPS*" function. Subtracting the yellow part of the graph would therefore give the indication of highest possible FPS, which is easily higher than 200, as is evident from the white lines marked with 100 and 200 fps respectively. The small yellow number to the left is the current amount of time spent idle, on the leftmost frame (the one marked with a white line).

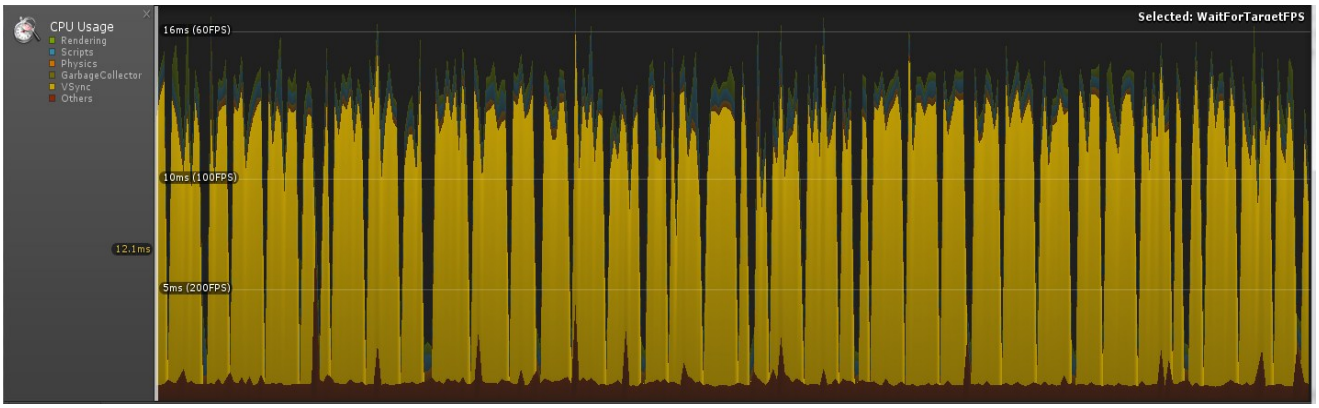


Figure 32 - 300 frames of profiling, WaitForTargetFPS highlighted in yellow.

From the performance tests described here, it can be argued that the performance of the prototype implementation is clearly sufficient to prove its applicability to real time applications, which is essential for any simulation or video game. It can be argued that more advanced test scenarios should be devised, the fact that conflict was generated from an even smaller scenario than this one, it can be said to be sufficient.

7.6 Discussion

The prototype presented in this chapter was made to show the potential of FIF to address the problem statement presented in section 1.2. The most prevalent question that this thesis seeks to address is the possibility of creating dynamic gameplay scenarios, with the added requirement of allowing scripted scenarios to drive narrative while still having the FIF produce the "filler content".

As to the most essential question, it must be said that the FIF is well capable of being the engine for dynamic gameplay generation through simulation of conflict between agents. This

is well supported by the simple example provided in this chapter. The additional problem of blending the dynamic system with more heavily scripted content is a different matter altogether. While the prototype does not utilize scripted content to any degree, it should be noted that we found it reasonably easy to manipulate the choices of agents by simply adjusting the weighting of attributes in the reasoning engine. This is arguably a poor argument for a more complex solution, as it would be increasingly complex to manage the relationship between the different values and the desire groups chosen by actors. The implication of this fact is that the FIF can be said to lend itself poorly, in its current incarnation, to seamless blending between scripted events and dynamic generation of gameplay.

In regards to the issue of scalability, the data presented in section 7.5 speaks for itself. The computational intensity of FIF in small scenarios is miniscule. However, as mentioned earlier, the possibility of combinatory explosions when presented with large game worlds is a real one. As the current version of the FIF will evaluate every single smart object in the entire world, capable of providing the desire value selected, as well as all attributes in the desire set, it is clear that this will quickly increase the computational requirements. There are however, several ways of addressing these issues. These are discussed in section 7.6.1.

As explained in section 1.2.1, there were several limitations imposed on the FIF design, to maintain the focus on proving viability, rather than implementing all the topics covered by Medler et al. [22]. These features are discussed further in 7.6.2 and 7.6.3.

7.6.1 Parallelism

The current implementation of the FIF was designed to run on a single thread, at the same pace as the game engine. This turned out to work well with the very small prototype used to test the FIF implementation, as described in chapter 7. However, the FIF was made for simulating much larger virtual worlds than the small system made to prove its viability. A very interesting direction for further research would be to investigate potential for parallelism in the system. There are several ways this could be accomplished; the most straight forward solution would simply be to run the entire FIF in a separate thread from the game engine. This should not be a problem as it is quite common for AI systems to run asynchronous from the remaining game engine [74].

A more interesting approach, would be to investigate the viability of running each actor in a separate thread, or as a part of a task based threading system. A task based system could even run with a finer granularity, running all attribute evaluations of an actor at the same time, and then evaluating potential services as the next step. This would allow a single actor to be distributed over an amount of threads equal to the smallest value of available threads an available attributes to evaluate.

As a final approach to parallelism, one could potentially move the entire FIF to a separate platform altogether. As the FIF runs as a separate layer, only loosely integrated with the game engine, it could be possible to have the FIF running on a separate platform, communicating with the rest of the simulation over some kind of network. This could be very interesting, for instance in building massively online multiplayer games (MMOGs) or large simulations of populations, for instance entire nations.

7.6.2 Advanced actor hierarchies

In their paper, Medler et al. [22] proposes an actor hierarchy where actors are arranged in hierarchies and form "groups". These groups can have different interests, and enable different behavior capabilities due to their focus. For instance, an actor could form a group with focus on getting access to more behavior strategies, for instance by training the members of the group in use of weaponry and acquiring this resource. Also, actors would have their desire values modified by their sub actors, causing actions of sub actors to propagate up the hierarchies.

These complexities in the actor implementation were left out to allow for the FIF prototype to be more easily constructed. Implementing these features would allow for highly complex interactions between actors, which in turn could potentially lead to a wider range of dynamic game scenarios being generated.

7.6.3 Solidarity

Solidarity is another concept presented in the paper by Medler et al [22]. This concept describes how actors would change their attitude towards parent actors, and their willingness to perform requests delegated to them. Actors would gain instability as the prestige attributes of their parent actor different greatly from their own, as well as the power of their parent no

longer affecting them in any meaningful way. This would cause actors to leave the hierarchy they were a part of, potentially joining other factions within the virtual world. Again this would allow for more complex faction interactions and a broader spectrum of potential gameplay scenarios; however it would also present a new challenge in regards to balancing the attributes of actors in such a way that the system did not drift apart.

Solidarity as a dynamic representation of the loyalty shown to parent actors would make for a very interesting addition to the FIF, however it would have to be a feature that could be turned on or off, as it is highly unlikely that all simulations would be improved by having factions break apart without the designers specifically intending for them to do so. Solidarity was not implemented in the prototype of FIF both due to the added complexity, and the potential for instability that such a feature would present. Even so, it would be worthwhile investigating the potential of such a feature for creating novel gameplay scenarios.

7.7 Summary

It turned out to be surprisingly easy to create a scenario, when making use of the FIF, where conflict is dynamically generated. Given only slightly different input parameters, the different actors make varied recommendations, causing each conflict to be slightly different. Given a more complex scenario, there is no doubt that even more varied scenarios would be generated. In this regard, there is no doubt that the FIF can be said to be performing admirably.

The small prototype presented in this chapter proves the viability of the implementation both in terms of real time performance and its ability to generate conflict scenarios as proposed in the design, this is supported by the analysis of the implementation as is described in section 7.4. Developers making use of the framework can define their world in terms of possible actions and attributes that are of interest to the actors of the world. By doing so, they can simply set the FIF to generate new events without having to script any sort of static gameplay.

While the prototype produced here only uses a small set of attributes, actors and smart objects, it is still a reasonable indicator of the framework's potential benefit to developers looking for a way to cheaply add more content to their open world games or simulations. It should of course be mentioned that the small prototype created here does not truly represent the complex game scenarios that one can find in modern games, such as the ones discussed in section 3.7. However, creating the game mechanics of such a virtual world is still as complex

as ever, even though the generation of gameplay scenarios has been automated by the FIF. This makes it impractical to build any large scale tests for a single research project. The implication of this is discussed further in chapter 8.

Chapter 8 Conclusion

This thesis has proposed a framework for how dynamic gameplay can be generated through the simulation of the interaction between factions. By providing scaffolding for the reader in both game development terminology and design principles, as well as introducing game engines that assist in this endeavor. In this chapter, a brief summary of all topics covered in this thesis is given in section 8.1 Section 8.2 describes, in more detail, the contributions made by this work. Finally, some thoughts are given in regards to future work in relation to the research presented here.

8.1 Summary

In this thesis, a possible approach to creating dynamic gameplay content through the simulation of faction interaction has been presented. This concept was spurned by the interest in answering the question "*How can one create additional gameplay content by simulating the interactions between the agents that populate a game world, while still allowing for scripted events to drive narrative?",* which was presented in section 1.2.

By investigating potential solutions in the field of game AI, a strategy was devised to build a generic framework capable of directing the flow of a virtual world through suggested applications of game mechanics. This framework, named the "*Faction Interaction Framework"* or FIF, made use of concepts from game and academic AI such as crowd simulation techniques, agent models, neural networks, fuzzy logic and smart objects. The core design of the FIF was presented as a whole in chapter 5.

To test the FIF in a reasonably realistic setting, an investigation was made into potential game engines that would allow for rapid integration of the FIF, as well as facilitating the creation of a small prototype. The evaluation of game engines was presented in chapter 4. The development language of choice and the implementation details of the FIF were presented in chapter 6. Finally, a prototype was presented in chapter 7.

In this prototype, it was shown how the FIF successfully generated dynamic gameplay scenarios, depending on the desire values, actions and behaviors available to the system. In addition, chapter 7 presented a discussion on the viability of the FIF and its potential for

solving the problem statement presented in section 1.2. Section 7.4 provided an analysis of the gameplay scenario created from the example presented in section 1.6; here it was shown that through simple manipulation of the desires of actors, new gameplay scenarios naturally emerge from faction interaction. Finally, an argument for the applicability to more complex virtual worlds was presented. Section 8.3.4 and 8.3.5 expand on this discussion with ideas for future work in regards to scalability.

8.2 Contributions

The FIF presents a viable design approach for creating emergent gameplay through modeling of factions and group interaction as a basis for conflict generation. By introducing this "additional layer" of reasoning to the virtual world, game developers can focus on creating interesting game mechanics while the gameplay is driven by the FIF recommendations.

As a side benefit, this thesis presents a set of game engines that can be used for quickly prototyping game mechanics and testing AI concepts. These engines are rated in regards of the functionality they provide, in addition to a performance test of the path finding and crowd simulation API of Unity3D.

Finally, this thesis provides an introduction to several advanced AI concepts. Some of these concepts are already being used in games, while others are mostly in the realm of academic AI as it stands. Hopefully, the game context of this thesis will allow others to see the advantages of these AI concepts in regards to game development.

8.3 Future work

Trying to create solutions for game development is immensely time consuming, as evident by the huge development costs of high end games, as discussed in section 1.2. The FIF appears to have great potential for alleviating this problem somewhat, by allowing for dynamic generation of content. Future work with dynamic content generation would entail investigating good implementations for more dynamic actor structures and actor solidarity, as discussed in section 7.6. In addition, as the reasoning engine used by actors in the FIF are based on neural networks, it would be interesting to investigate the potential for making use of other aspects of learning AI.

Learning systems has been the focus of many research and development projects in games studios the later years [74], but has proven to be mostly unsuccessful. Even so, learning systems is a widely researched topic in academic AI, and has been proven to work very well in various applied fields, such as translation, speech recognition and more [15]. Given the close connection between the actor reasoning engine core and neural networks (which can benefit from learning), investigating the potential for creating training sets that adjust attribute weights could prove beneficial. By doing this, one could potentially manipulate actors into following a distinct pattern of recommendations that would allow the world to evolve in a predictable fashion, when this is desirable.

References

[1] D. Ingie, "How a man is created," man.png, ed., Carnegie-Mellon's Entertainment Technology Center., 2009.

[2] ESRB. "Video Game Industry Statistics," 30. June, 2012; http://www.esrb.org/about/video-game-industry-statistics.jsp.

[3] M. Sharkey. "Report: Game Development Costs Have Skyrocketed," 30. June, 2012; http://www.gamespy.com/articles/108/1082176p1.html.

[4] Y. Takatsuki. "Cost headache for game developers " 2011; http://news.bbc.co.uk/2/hi/business/7151961.stm.

[5] C. Esmurdoc, "Postmorten: Double Fine's Brutal Legend," *Game Developer Magazine*, no. 12, December, 2009.

[6] N. Motion. "Euphoria," 7. October, 2011; http://www.naturalmotion.com/euphoria.

[7] Havok, "Havok AI Efficient Pathfinding for Dynamic Game Environments," *www.havok.com*, H. Inc, ed., Havok Inc, 2011.

[8] R. D. Smith, "Essential techniques for military modeling and simulation," in Proceedings of the 30th conference on Winter simulation, Washington, D.C., United States, 1998, pp. 805-812.

[9] Kongsberg-Group. "Simulator-kontrakt med den svenske hæren," 24. August, 2011; http://www.kongsberg.com/nb-no/kog/news/2010/november/1811_ctc-gbad_contract/.

[10] A. Pope, P. Selfridge, and J. R. Surdu, "Realistic agent populations for large-scale virtual training environments," in Proceedings of the 2008 Spring simulation multiconference, Ottawa, Canada, 2008, pp. 745-751.

[11] J. Gratch, and S. Marsella, "Tears and fears: modeling emotions and emotional behaviors in synthetic agents," in Proceedings of the fifth international conference on Autonomous agents, Montreal, Quebec, Canada, 2001, pp. 278-285.

[12] R. Casais, "Challenges with distributed systems at Funcom," University of Oslo

Funcom, 2011.

[13] V. Corp. "Left 4 Dead," 30. June, 2012; http://www.valvesoftware.com/games/l4d.html.

[14] M. Booth, "AI systems of L4D," in Artificial Intelligence and Interactive Digital Entertainment Conference, Stanford, 2009, pp. 95.

[15] A. Russel, and P. Norvig, *Artificial Intelligence a Modern Approach*, 3 ed.: Pearson Education, 2010.

[16] Bethesda. "The Elder Scrolls: Skyrim," 30. June, 2012; http://www.elderscrolls.com/skyrim/.

[17] E. Gilbert. "For PS4 And Xbox 720, It's Better AI, Not Graphics, Ubisoft Wants," 1. July, 2012; http://www.gamesthirst.com/2011/07/06/for-ps4-and-xbox-720-its-better-ai-not-graphics-ubisoft-wants/.

[18] K. Baldwin. "So you're "not ready" for next-gen consoles? Here's why you should be," 1. July, 2012; http://www.gamejudgment.com/so-youre-not-ready-for-next-gen-consoles-heres-why-you-should-be.

[19] G. N. Yannakakis, “Game AI revisited,” in Proceedings of the 9th conference on Computing Frontiers, Cagliari, Italy, 2012, pp. 285-292.

[20] S.-P. v. Houten, and A. Verbraeck, “Controlling simulation games through rule-based scenarios,” in Proceedings of the 38th conference on Winter simulation, Monterey, California, 2006, pp. 2261-2269.

[21] R. Abbott, “Emergence explained: Abstractions: Getting epiphenomena to do real work: Essays and Commentaries,” *Complex.,* vol. 12, no. 1, pp. 13-26, 2006.

[22] B. Medler, J. Fitzgerald, and B. Magerko, “Using conflict theory to model complex societal interactions,” in Proceedings of the 2008 Conference on Future Play: Research, Play, Share, Toronto, Ontario, Canada, 2008, pp. 65-72.

[23] D. E. Comer, D. Gries, M. C. Mulder *et al.*, “Computing as a discipline,” *Commun. ACM,* vol. 32, no. 1, pp. 9-23, 1989.

[24] I. Millington, and J. Funge, *Artificial Intelligence for games*, 2 ed.: Morgan Kaufmann, 2009.

[25] J. Gregory, *Game Engine Architecture*, Natick: A K Peters, 2009.

[26] A. Thorn, *Game Engine Design and Implementation*, London: Jones and Bartlett Learning, 2011.

[27] Carnegie-Mellon, "Panda3D," Carnegie Mellon Entertainment Technology Center, 2010, p. Game Engine.

[28] UnityTechnologies. "Unity3D Manual," 01.06.2012, 2012; http://unity3d.com/support/documentation/Manual/index.html.

[29] D. Livingstone, “Turing's test and believable AI in games,” *Comput. Entertain.,* vol. 4, no. 1, pp. 6, 2006.

[30] T. Akenine-Möller, E. Haines, and N. Hoffman, *Realtime Rendering*, 3 ed.: A K Peters, 2008.

[31] D. Fu, and R. Houlette, "The Ultimate Guide to FSMs in Games," *AI Game Programming Wisdom 2*, AI Game Programming Wisdom S. Rabin, ed., p. 38, 2004.

[32] A. J. Champandard. "Behavior Trees for Next-Gen Game AI," 2011; https://aigamedev.com/insider/presentations/behavior-trees/.

[33] M. Buckland, "Programming game AI by example," p. 44, Sudbury, MA: Wordware Publishing, 2005.

[34] C. Bailey, and M. Katchabaw, “An emergent framework for realistic psychosocial behaviour in non player characters,” in Proceedings of the 2008 Conference on Future Play: Research, Play, Share, Toronto, Ontario, Canada, 2008, pp. 17-24.

[35] A. Braun, B. E. J. Bodmann, and S. R. Musse, “Simulating virtual crowds in emergency situations,” in Proceedings of the ACM symposium on Virtual reality software and technology, Monterey, CA, USA, 2005, pp. 244-252.

[36] S. Zhou, D. Chen, W. Cai *et al.*, “Crowd modeling and simulation technologies,” *ACM Trans. Model. Comput. Simul.,* vol. 20, no. 4, pp. 1-35, 2010.

[37] A. Russell, "Turning spaces into places," *AI game programming wisdom 4*, Ai game programming wisdom S. Rabin, ed.: Course Technology, 2008.

[38] T. Abaci, J. Ciger, and D. Thalmann, “Planning with Smart Objects,” in WSCG Plzen, Czech Republic, 2005, pp. 4.

[39] M. Kallmann, and D. Thalmann, “A Behavioral Interface to Simulate Agent-Object Interactions in Real Time,” in Proceedings of the Computer Animation, 1999, pp. 138.

[40] A. J. Champandard. "Living with the sims' ai: 21 tricks to adopt for your game," 24.07.2011, 2011; http://aigamedev.com/open/highlights/the-sims-ai/.

[41] Maxis, "The Sims," Electronic Arts, 2000.

[42] P. Sequeira, M. Vala, and A. Paiva, “What can i do with this?: finding possible interactions between characters and objects,” in Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems, Honolulu, Hawaii, 2007, pp. 1-7.

[43] L. M. G. Goncalves, M. Kallmann, and D. Thalmann, “Programming Behaviors with Local Perception and Smart Objects: An Approach to Solve Autonomous Agents Tasks,” in Proceedings of the XIV Brazilian Symposium on Computer Graphics and Image Processing, 2001, pp. 184.

[44] L. Dicken, "HCSM: A Framework for Behavior and Scenario Control in Virtual Environments," http://aigamedev.com/open/review/hcsm-concurrent-state-machine/, [10. October, 2009].

[45] T. Tutenel, R. Bidarra, R. M. Smelik *et al.*, “The role of semantics in games and simulations,” *Comput. Entertain.,* vol. 6, no. 4, pp. 1-35, 2008.

[46] A. Sud, E. Andersen, S. Curtis *et al.*, “Real-time path planning for virtual agents in dynamic environments,” in ACM SIGGRAPH 2008 classes, Los Angeles, California, 2008, pp. 1-9.

[47] M. Kallmann, “Navigation queries from triangular meshes,” in Proceedings of the Third international conference on Motion in games, Utrecht, The Netherlands, 2010, pp. 230-241.

[48] P. Marden, F. Smith, C. Mcanlis *et al.*, "Movement and Pathfinding," *AI Game Programming Wisdom*, 4, S. Rabin, ed., p. 144, Boston: Course Technology, Cengage LEarning, 2008.

[49] M. Booth. "Source Engine Navigation meshes," 10.01, 2012; https://developer.valvesoftware.com/wiki/Navigation_Meshes.

[50] T. C. H. John, E. C. Prakash, and N. S. Chaudhari, “Strategic team AI path plans: probabilistic pathfinding,” *Int. J. Comput. Games Technol.,* vol. 2008, pp. 1-6, 2008.

[51] D. Harabor, and A. Grastien, “Online Graph Pruning for Pathfinding on Grid Maps,” in AAAI Conference on Artificial Intelligence (AAAI), San Fransisco, USA, 2011.

[52] E. Games. "Unreal Technology," 3. July, 2012.

[53] Crytek. "CryEngine 3," 11. October, 2011; http://www.crytek.com/cryengine.

[54] S. V.Kharkar, "AI Waterfall: Populating Large Worlds Using Limited Resources," *AI Game Programming Wisdom*, S. Rabin, ed., p. 9: Thomson Delmar Learning, 2006.

[55] B. Sunshine-Hill, and N. I. Badler, “Perceptually Realistic Behavior through Alibi Generation,” in Conference on Artificial Intelligence and Interactive Digital Entertainment, Stanford, California, USA, 2010.

[56] M. Buro. "The 2nd Annual AIIDE Starcraft AI Competition," 09.01.2012, 2012; http://StarcraftAICompetition.com.

[57] springrts. "Spring Engine," http://springrts.com/.

[58] A. Heinermann, and "Kovarex". "BWAPI," 09.01, 2012; http://code.google.com/p/bwapi/.

[59] GameMiddleware.org. "Middleware," 7. October, 2011; http://www.gamemiddleware.org/middleware.

[60] G. Games", "Torque3D," GarageGames, 2011.

[61] Microsoft, "XNA Game Studio," Microsoft, 2010.

[62] Havok. "Solutions for Engineers," 07. October, 2011; http://www.havok.com/solutions/game-engineers.

[63] UnityTechnologies. "3.5 Developer Preview Public Beta Release Notes," 13.02, 2012.

[64] M. Mononen. "recastnavigation," 01.09, 2012; http://code.google.com/p/recastnavigation/.

[65] Microsoft. "Game.TargetELapsedTime property," 29. July, 2012; http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.game.targetelapsedtime.aspx.

[66] J. Kessing, T. Tutenel, and R. Bidarra, "Services in Game Worlds: A Semantic Approach to Improve Object Interaction," in Proceedings of the 8th International Conference on Entertainment Computing, Paris, France, 2009, pp. 276-281.

[67] F. Lamarche, and S. Donikian, "Automatic orchestration of behaviours through the management of resources and priority levels," in Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 3, Bologna, Italy, 2002, pp. 1309-1316.

[68] Xamarin. "About Mono," 14.11, 2011; http://www.mono-project.com/About.

[69] S. Ryazanov. "The Impossibly Fast C++ Delegates," 09.06, 2012; http://www.codeproject.com/Articles/11015/The-Impossibly-Fast-C-Delegates.

[70] Microsoft. "Events and Delegates," http://msdn.microsoft.com/en-us/library/17sde2xt(v=vs.90).aspx.

[71] Microsoft. "internal (C# Reference)," 09.06, 2012; http://msdn.microsoft.com/en-us/library/7c5ka91b.aspx.

[72] Microsoft. "Properties (C# Programming Guide)," http://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx.

[73] UnityTechnologies. "Profiler (Pro only)," 8. July, 2012; http://docs.unity3d.com/Documentation/Manual/Profiler.html.

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