OULUN AMMATTIKORKEAKOULU

Ilpo Alatalo

NEED-BASED ARTIFICIAL INTELLIGENCE FOR A COMPUTER GAME

Case Spaceship Captain –prototype

NEED-BASED ARTIFICIAL INTELLIGENCE FOR A COMPUTER GAME

Case Spaceship Captain –prototype

Ilpo Alatalo
Thesis
Autumn 2014
Business Information Technology
Oulu University of Applied Sciences

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietojenkäsittely, web-sovelluskehitys

Tekijä(t): Ilpo Alatalo
Opinnäytetyön nimi: Tarve-pohjainen tekoäly tietokone peliä varten
Työn ohjaaja: Ani Ruusila
Työn valmistumislukukausi- ja vuosi: syksy 2014
Sivumäärä: sivut + liitteet: 29 + 2

Tämän opinnäytetyö tavoitteena oli kehittää tarve-pohjaisen tekoälyn prototyyppi tietokonepeliä varten. Työlle ei ollut kolmannen osapuolen toimeksiantoa vaan tein sen omasta mielenkiinnostani sekä edistämään tietotaitoani.

Olen kehittänyt Spaceship Captain nimistä peliä jo jonkin aikaa ja opinnäytetyön tekoäly kehitettiin sitä varten. Spaceship Captain on tarinavetoinen 2D-elämäsimulaatio-/roopeli PC:lle ja sen pääominaisuuksina ovat sen satunnaisgeneroidut hahmot sekä näiden The Sims -pelejä muistuttava sosiaalinen tekoäly. Koska pelin tekoäly kokonaisuudessaan olisi ollut liian laaja ja monimutkainen aihe opinnäytetyölle, päätin keskittyä sen sijaan vain sen tarve-pohjaiseen toiminnallisuuteen.

Prototyyppi tehtiin käyttäen Unitya ja Microsoft Visual Studiota. Tärkeimmät metodit prototyypin kehityksessä olivat ääreelliset automaatit, sumea logiikka, navigaatioverkot ja älyesineet. Koska The Sims -pelien tekoäly on myös tarve-pohjainen, käytin sitä lähtokohtana tekoälyn kehitykselle. En kuitenkaan ottanut tarpeita The Sims -peleistä, vaan käytin sen sijaan Maslown tarvehierarkiaa pohjana ja muokkasin sitä omien vaatimuksieni mukaiseksi.

Tekoälyn prototyyppi kehitettiin onnistuneesti, vaikkakaan siitä ei tullutkaan niin laajan kuin alunperin suunnittelin. Vain kourallinen suunnitelluista tarpeista päätyi prototyyppiin, mutta tämä ei toisaalta ollut ongelma, sillä puuttuvien tarpeiden toteuttaminen halutulla tavalla olisi vaatinut lisäjärjestelmien tekemistä prototyyppiin. Tarvepohjaisuuden peruslogiikka toteutettiin kuitenkin onnistuneesti.

Opinnäytetyön kirjoittamisen aikana huomasin muutamia parannuksen kohteita prototyyppiin, mutta laajuutensa vuoksi jätin ne jatkokehitystä varten.

Asiasanat: peli, tietokonepeli, tekoäly, Unity, The Sims

ABSTRACT

Oulu University of Applied Sciences
Business Information Technology, web-application development

Author(s): Ilpo Alatalo
Title of thesis: Need-based artificial intelligence for a computer game
Supervisor(s): Ani Ruusila
Term and year when the thesis was submitted: autumn 2014
Number of pages: 29 + 2

The goal of this thesis was to develop a prototype of a need-based artificial intelligence (AI) for a computer game. The thesis wasn't commissioned by a third party, but was done out of my own interest and to improve my know-how.

The AI was developed for Spaceship Captain, a game I have been working on for some time. Spaceship Captain is a story focused, 2D-life simulation/role-playing game for PC. The game's central feature is random generated non-playable characters and their social AI, highly reminiscent of the AI seen in The Sims. As the complete AI was going to be too complex subject for a thesis, I instead decided to only cover the need-based functionality.

The prototype was developed with Unity and Microsoft Visual Studio. The most important methods for developing the prototype were finite state machines, fuzzy logic, navigation meshes and smart objects. The AI of The Sims was used as a jumping-off as it's also need-based. Instead of taking the needs that The Sims uses though, Maslow's hierarchy of needs was used a base instead and later modified further.

The AI prototype was successfully developed, though not as expansively as first envisioned. Only a handful of the designed needs made their way into the prototype, but this ultimately wasn't a real issue, as implementing all of the needs as designed, they would've needed additional systems to accompany them. The core of the need-based functionality was however developed as planned.

During the writing of this thesis, a handful of points of improvement were discovered with the prototype, but as again they weren't something to be dealt with within the scope of the thesis, they were left for future development.

Keywords: game, computer game, AI, artificial intelligence, Unity, The Sims

TABLE OF CONTENTS

1 INTRODUCTION

This thesis is about the design and development of an artificial intelligence (AI) for a computer game. The subject was not commissioned by a third-party, but was chosen by me out of my own interest and use for my own projects.

For some time now I have been working on a computer game project, work name “Spaceship Captain”. On a technical level, the game is built around a sophisticated, need-based, social AI, somewhat similar to the AI found in the game series The Sims. As the scope of Spaceship Captain is large and the technical side of its development complex, I decided to start the development of the game from the AI, as it's a core feature of the game, around which the rest is built upon. The scope of this thesis was to design and develop a prototype of the need-based functionality of the AI. Further functionality was considered, but ultimately left out in order to keep the thesis' scope manageable.

As far as the game Spaceship Captain is concerned, everything described in this thesis is subject to change. As the design of the game and the AI is not yet final, this thesis may or may not end up reflect the final product, if and when it is complete.

2 ABOUT SPACESHIP CAPTAIN

Spaceship Captain is a story focused 2D –life simulation/role-playing game for PC in which the player steps in to the shoes of a captain of a space-transport ship. The player's goal in the game is to transport people and goods around the galaxy, whilst maintaining the safety and operability of himself, his ship and his passengers.

The game takes place in a sci-fi universe, alternative to our own, far in the future. Despite its strong narrative focus, the game doesn't follow a traditional story. The game's world is designed to rich with detail and fully dynamic, leaning on the use of emergent storytelling instead of a linear narrative. The story is shaped by the decisions made by the player, making it different on every playthrough.

Mechanically Spaceship Captain is a light mix of role-playing and management. The player uses most of his/her time controlling the player character around the ship, interacting with objects and speaking to other characters. The game is viewed from a top-down perspective and is mainly controlled with a mouse. The game's central themes are survival and morality and the player is constantly put into situations, where they must weigh the value of their own life and the lives of others and also the morality of their choices. The game doesn't have a definite end, but goes on until the player character dies or his/her ship is destroyed.

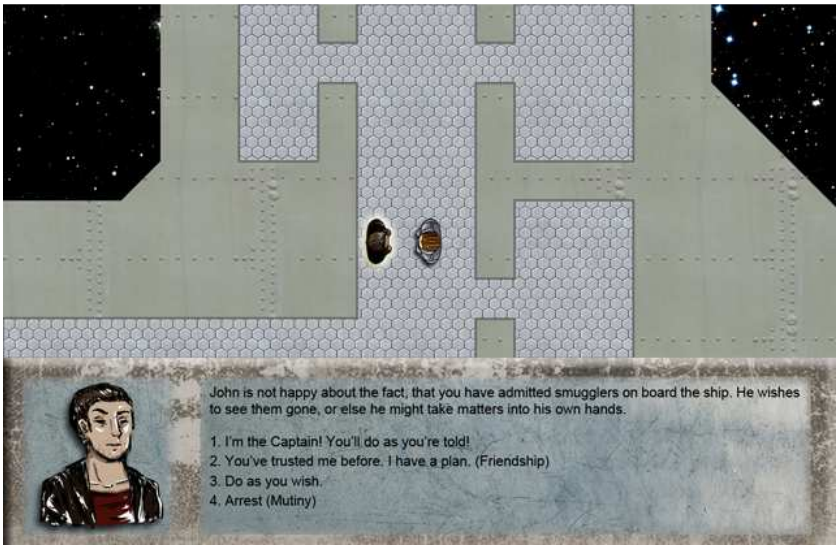

FIGURE 1. A concept picture of how the game might look

The central feature of the game is its random generated non-playable characters (NPC) and their social AI. Because the narrative of the game revolves almost entirely around character interaction, the AI has to be able to create the illusion of characters with unique personalities, habits and histories. To this end the AI is designed to be fully autonomic, able to take care of itself and interact not only with the player, but also other AI characters. A big part of this is the dialogue system, through which most of the character interactions happen. When talking to characters, the player has a predetermined pool of topics and responses that grow and change depending on the state of the relationship the player has with the character. The characters also remember what they've told the player before and what the player in change has told them. Though technically quite simple, the characters persistent memory of earlier conversations makes the system feel more realistic and believable.

3 DEVELOPMENT TOOLS

3.1 Unity

The main development tool for Spaceship Captain is Unity (also known as Unity 3D) which is at the time of writing the most popular game development software globally (Unity Technologies 2014a, referenced 19.11.2014). Unity is a game development ecosystem, meaning it includes both a game/rendering engine and an integrated development environment (IDE), thus making it possible to create full video games with no additional software, excluding the creation of the game's assets (3D-models, sounds, textures etc.). Unity was originally made for the creation of 3D games, but it has grown to support the creation of 2D games with the addition of internal 2D development tools (Unity Technologies 2013, referenced 19.11.2014). Unity supports cross-platform development, meaning a game project can be built for multiple platforms. Supported platforms include Windows, Mac, Linux, mobile platforms, web browsers and consoles. (Unity Technologies 2014b, referenced 19.11.2014.)

The project was made with the free version of Unity. Additional features such as better cross-platform support, greater graphics tools and options can be unlocked with the paid Pro-version license, but for the scope of the project these weren't necessary (Unity Technologies 2014c, referenced 19.11.2014).

3.2 Microsoft Visual Studio

For development Unity supports three programming languages: C#, JavaScript (also called UnityScript) and Boo. Unity comes bundled with a code editor called MonoDevelop, but external editors can also be used. (Unity Technologies 2014d, referenced 19.11.2014.)

Thought the MonoDevelop is perfectly adequate for Unity's programming needs, out of personal preference the code editor of choice for Spaceship Captain was Microsoft Visual Studio 2013. Visual Studio is an IDE, designed for developing applications for desktop, web and cloud devices (Microsoft 2014a, referenced 2.12.2014). It comes in many different versions, such as Ultimate,

Premium and Professional, each with a different set of features, but the version used in this project was Express (Microsoft 2014b, referenced 2.12.2014).

The reason I chose Express over the other versions was primarily because of the cost: Express is free where the other version are not. Also, since I only needed a lightweight and fast code editor, Express suited my needs perfectly. Being a free version, Express is quite lightweight since it doesn't have many unneeded features to weigh it down.

4 ARTIFICIAL INTELLIGENCE METHODS

4.1 Finite State Machine

Finite state machines (FSM) are one of the most important tools in an AI-programmer's tool belt. In its most simple form - as its name suggests - a FSM is in charge of changing the state of an AI-character from a set of predetermined set of states. The number of possible states can vary greatly depending on the need of the AI, but the set of states doesn't change during execution. The conditions leading to the states however, can change. (Mark 2009, 30.)

In a first-person shooter game for example a FSM could be in charge of changing an enemy's state from idle to attacking to fleeing to dying. The enemy might start in an idle state and it will stay in that state until one of the state changing conditions is met. If the player for example enters a certain radius of the enemy or the enemy establishes line-of-sight with the player, its state would change to attacking. If the player manages to reduce the enemy's health to 10% of its maximum, its state would change to fleeing. Lastly, if the enemy's health reaches zero, its state changes into dying. (Mark 2009, 30.)

Naturally an enemy like this would be highly predictable and not smart at all. The player could easily avoid the enemy altogether or the enemy might flee from the player indefinitely if it had no way of regaining health. By applying algorithms and complex conditions to state changes, even a simple FSM like this could make an enemy seem smarter. (Mark 2009, 30.)

4.2 Fuzzy Logic

Fuzzy logic in the simplest terms means turning a binary decision into non-binary. If your aim is to make a "believable" AI, fuzzy logic is one of the best tools for the job. As an example lets use a hypothetical role-playing game (RPG) in which the combat mechanics consist of three types of attacks: melee, spells and ranged. Let’s say each of our attack types has an ideal range: melee is close range, spells are mid-range and ranged is long range. Our AI needs to be able to change its attack style depending on the player’s actions, so naturally using the distance between the player and the AI as the condition for changing our type of attack seems like the smart thing to do. But

just like before, the AI is completely predictable and thus doesn't seem smart or believable at all. (Mark 2009, 31-32.)

Now we could make the conditions more complex with multiple terms, but for a simpler solution we can just use fuzzy logic. Instead of a binary choice, we use the distance between the player and the AI and apply it to a simple arithmetic function, turning it into a probability. This translates into a more unpredictable and so more believable AI. At close range the AI is most likely to use a melee attack, but it is also quite likely to cast a spell and even though it might not be the best course of action, the AI could also use a ranged attack, which might throw the player off balance. (Mark 2009, 31-32.)

4.3 Navigation mesh

Navigation meshes (short: navmesh), like their name suggests, are a method used by AI's to navigate a virtual space. In the past the primary method of navigation for AIs was to use waypoints - singular points placed around the environment, between which the AI-character would move. While waypoints work quite fine in a simple, controlled environment, they creates a plethora of problems when placed into a large, complex and dynamic environment. Navmeshes on the other hand are more suited for such conditions. (PaulT 2008, referenced 2.12.2014.)

A navmesh is as its name states, a mesh: a collection of polygons (nodes) linked together to form a mesh. Important to note however, is the fact that the polygons that form the mesh must be convex polygons (as opposed to concave polygons). In a convex polygon all of the interior angles are less than 180 degrees (Epic Games 2012. Referenced 2.12.2014).

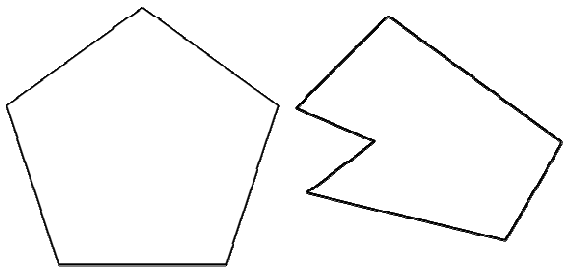

FIGURE 2. A convex polygon (left) and concave polygon (right)

The need for convex polygons is integral to the functionality of a navmesh. Inside a convex shape, a line can be drawn between any two points, which in turn makes it possible to construct a path from any two points inside a navmesh, by drawing lines between the nodes. A navmesh usually has to be different from the graphical mesh of the terrain, because calculating movement across a high-detail mesh is too computer intensive. For this reason the navmesh is either specifically made by a 3D-artist or computer calculated (aka. baked) from the graphical mesh. (Epic Games 2012. Referenced 2.12.2014).

FIGURE 3. Waypoint navigation (left) vs. navmesh navigation (right) (PaulT 2008. Referenced 2.12.2014)

4.4 Smart Objects

The term “Smart Object” in the context of this thesis originates from The Sims -series. As the name strongly suggests, smart objects are objects with intelligence – namely they contain parts of the intelligence necessary for the AI characters to interact with said objects. In the case of The Sims, the objects would contain scripts, animations, navigation information and much more, effectively containing all the related AI data. (Champandard 2007. Referenced 2.12.2014.)

For The Sims, from a technical standpoint the choice to use smart objects in its AI was done for three reasons: editability, maintainability and ease of debugging. When working on a game with the scale of The Sims, it’s important that the tools in your disposal maximize productability. The game has hundreds (check?) of objects and though most are just “duplicates” of one another, not all are completely identical and even the actual individual “object types” are in the double digits. If every object or even every object type would have had to be programmed into the game individu-

ally, it would've been an enormous task. And if all the necessary AI logic was to be in a single place, for example the characters, editing, maintaining and debugging the code would've been extremely inefficient. So it would only make sense in the case of The Sims to have the AI's interaction logic in the object themselves. This also made adding new objects into the game effortless. (Champandard 2007. Referenced 2.12.2014.)

Another great feature of the smart objects is their ability to "advertise" the needs they sate. Though from a technical standpoint a very minor feature and almost too obvious to even mention, its importance only becomes more apparent when a character has multiple needs at once and has to decide which to sate next. Does the AI choose the most critical need and focus on that, or does it play smart and try to maximize its effectiveness by satisfying multiple needs at once. (Champandard 2007. Referenced 2.12.2014.)

5 REFERENCES

When designing and developing an AI of this scale and complexity, it's easy to get lost in the details and not see the big picture. Coming up with every solution on your own takes lots of time and considerable effort, so instead of reinventing the wheel, I decided it would be smart to refer to other sources to find help.

5.1 The Sims

Perhaps the biggest source of inspiration for Spaceship Captain, especially from the AI's point of view, has been The Sims -series and more closely the first game in the series. Though there have been lots of other games with exceptional AI, The Sims' resembles the type of functionality I'm looking for. It's also a very accessible and well documented game so it was easy to find detailed information about it.

My main focus when researching The Sims was the AI's need-based functionality and the associated smart objects, as they were a big part of the core functionality I wanted to have in my own AI. There were also other design and gameplay focused aspects of the game which heavily inspired me, but as they are not within the scope of this thesis, they shall be left undiscussed.

FIGURE 4. Motives as they appear in The Sims (Electronic Arts 2010, referenced 26.11.2014)

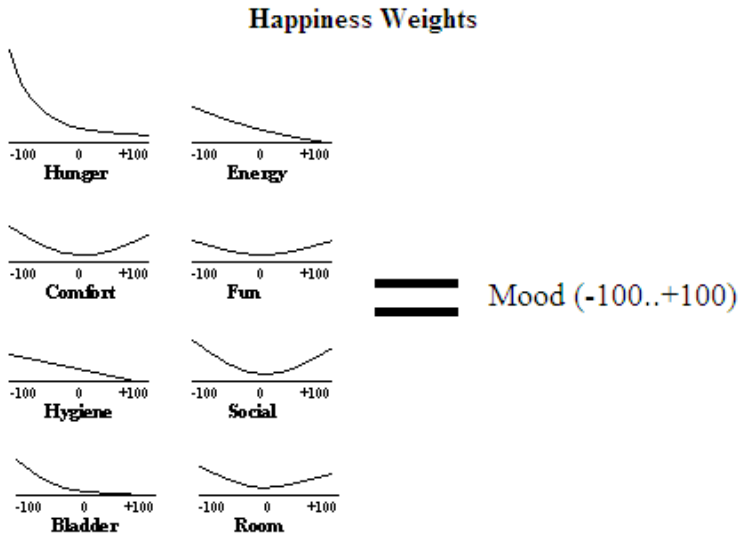

FIGURE 5. Happiness Weights in The Sims (AI Game Programmers Guild, referenced 16.12.2014)

The actual core logic of The Sims' AI's need-based nature is surprisingly simple: the AI only aims to maximize a character's happiness. Each of the motives, as seen in figure 5, affects the overall happiness of a Sim, each to a different degree. As stated in the previous chapter, all the objects in the game "advertise" the needs they sate, which can be multiple at once. If left to their own vices, the AI in The Sims choose the next object to interact with based on cumulative "happiness score" the object provides. *(AI Game Programmers Guild, referenced 16.12.2014.)*

5.2 Maslow's Hierarchy of Needs

Abraham Maslow (1908-1970) was a well known American psychologist, best known as being the father of Humanistic Psychology and his theory of the Hierarchy of Needs. Instead of focusing on the abnormalities of human nature like most psychologists of his time, Maslow was more interested in positive psychology like human potential and improving mental health. (Cherry 2014, referenced 22.12.2014.)

With the publication of his paper "A Theory of Human Motivation" in 1943 and later his book "Motivation and Personality" in 1954, Maslow theorized that people are motivated to achieve certain needs in a hierarchical model: when the lowest level of needs is fulfilled a person moves on to the next level, eventually working their way up to self-actualization. This model is Maslow's Hierarchy

of Needs. (Abraham Maslow 2009, referenced 22.12.2014; McLeod 2014, referenced 22.12.2014.)

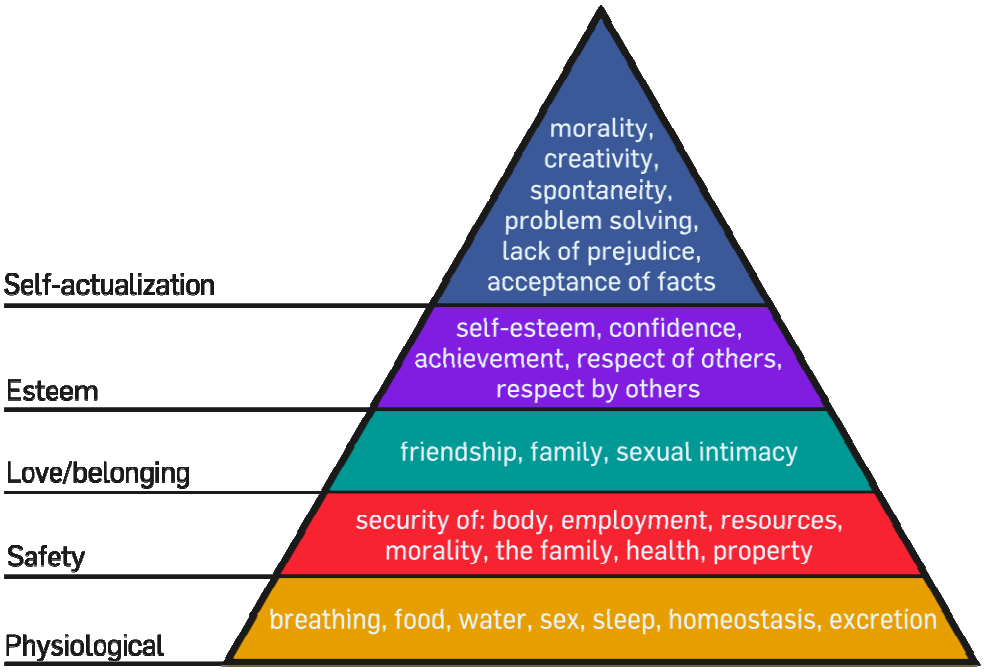

FIGURE 6. Maslow's Hierarchy of Needs (Factoryjoe 2009, referenced 26.11.2014)

The original hierarchy, as seen in figure 6, had five levels, but was later expanded upon by Maslow in the 1960's and 1970's to include three more levels: cognitive, aesthetic and transcendence needs. This expanded hierarchy however was not chosen for use in the development of AI for Spaceship Captain, because as discussed in more detail in the next chapter, not even the original hierarchy is fully utilized.

6 DESIGNING & DEVELOPING THE AI

6.1 Designing the needs

The first step in creating a need-based AI was defining the needs. As the AI in The Sims was very similar to what I was looking for, it was an excellent jumping-off point. Nevertheless, as Spaceship Captain is fundamentally a very different game from The Sims, using its needs "as is" wouldn't do - its characters were never intended to be fully autonomic, which was one of the core ideas in Spaceship Captain. The Sims AIs needs are very basic and they all work in the same fashion and two of the needs – Comfort and Room – are both heavily tied to the building and decorating aspects of the game.

I wanted Spaceship Captain's AI to be more realism and to that end after researching about the basic human needs. One of the core behaviors I wanted the AI to depict was to treat its needs unequally. As humans, we place survival above all else, which means when we are for example hungry or tired, we take care of those needs before others because they are paramount to our survival.

During my research I came across Maslow's Hierarchy of Needs and decided to utilize it almost straight away. It covered most of the needs I wanted to have in the game and its tier-based structure perfectly encompassed the priorization of certain needs above others I wanted. I eventually ended up focusing mainly on the bottom three tiers of the pyramid, as they encompassed more or less all the needs I wanted to include in the game. Another reason was my assessment that the higher tiers – thought they would've made the AI far more sophisticated – would've raised the difficulty of programming of the AI exponentially. I also believe they would've ultimately driven the project off course, because the aim of the project wasn't to create the world's most advanced AI, but one that suits the needs of the game. The hierarchy I finally ended up with is as follows:

- **Primary:** hunger, bladder, energy, health
- **Secondary:** hygiene, work
- **Tertiary:** entertainment, social, sex

The needs as they currently are, come in two variants: dynamic and static. Just like in real life, the basic human needs, like hunger and sleep have to be satisfied regularly in order for a person to stay functioning. These needs are dynamic and most of the needs in the game fall into this category. A few of the needs (currently health and work) behave differently. They are static, meaning they don't fluctuate constantly, but stay the same until affected by outside forces, like another character.

6.2 Scoring the needs

Before the AI can make any decisions, it needs to know how likely it is to make a choice. To this end all the needs of the AI must to be scored. In order to simulate the degradation of needs correctly, this is done automatically every frame by Unity's internal Update-function. Each of the needs has a value from 99 to 0, which when subtracted from the maximum serves as the base score. It is then multiplied by a multiplier, which is based on the needs tier and state. As discussed earlier, the needs are split into three tiers based on Maslow's hierarchy, but in addition to this, the needs also have three states: high, medium and low. The intervals for these states differ between needs.

```
65      public float GetScore() {
66          if (priority == 1) {
67              if (current >= high) score = (99 - current) * 0.5f;
68              else if (current >= mid) score = (99 - current) * 0.7f;
69              else score = (99 - current) * 1.0f;
70          }
71          else if (priority == 2) {
72              if (current >= high) score = (99 - current) * 0.6f;
73              else if (current >= mid) score = (99 - current) * 0.8f;
74              else score = (99 - current) * 0.9f;
75          }
76          else {
77              if (current >= high) score = (99 - current) * 0.8f;
78              else if (current >= mid) score = (99 - current) * 0.8f;
79              else score = (99 - current) * 0.8f;
80          }
81          return score;
82      }
```

FIGURE 7. Scoring function for the AI's needs

It is important that the score multiplier is affected by both the tier and state of the need, as otherwise the AI's behavior would be less realistic or would make it less suitable for a game. If determined only by the tier, the AI would spend most of its time sating the primary needs, which isn't

how modern humans behave. We eat, sleep and go to the toilet usually only when we feel it's necessary. On the flipside, if the multiplier was only dependent on the state of the needs, the whole tier based structure would serve no purpose and the AI might decide to watch TV and starve instead of thinking about survival like actual humans would. The differing state intervals also serve an important function. If the AI would be just as likely to sleep as to go to the toilet, it would not be believable.

6.3 Deciding and performing an action

```
78          if (actionState == 0) DecideNextAction();
79          else if (actionState == 1) FindNeedObject();
80          else if (actionState == 2) {
81              currentInteractionObj.SendMessage("StartUsing", gameObject);
82              actionState = 3;
83          }
```

FIGURE 8. Main loop of the character AI

The characters AI's core logic loop is handled by a simple finite state machine (FSM) with four states, controlled by a single variable "actionState". In the FSM's first state (actionState == 0), the AI starts deciding what need it needs to sate next. The decision making process starts by sorting all the needs by score and then limiting the final decision pool of needs by an arbitrary number, which in my case I chose to be four, so that the AI cannot choose the currently most sated need for example when there are more pressing needs to consider.

After the final decision pool is chosen, fuzzy logic is then used to determine what the final choice will be. Using basic probability calculus, each need and their "score range" is placed into an array and a random number from zero to the cumulative score of the needs is drawn and checked against the "score ranges" of each need. The need with the highest score has the biggest probability of being chosen, while the least scoring need has the lowest.

After the next need is decided, the FSM changes into the next state (actionState == 1), in which the AI finds the object that best sates the chosen need. Somewhat similarly to how the smart objects in The Sims "advertise" the needs they affect, in Unity game objects can be tagged. The AI lists all the game objects tagged with the chosen need and then finds the best game object based on a set of predetermined conditions.

First and foremost, no matter what object the character is looking for, the AI always checks if the object is already in use and filters out any objects currently in use. This functionality is still rather simple, as the prototype doesn't have multiple characters, but the functionality will be important in the future.

When looking for food (sating hunger), the AI tries to find the food object that most closely matches it's hunger level. This is done by arranging the food objects in descending order of "feeding power", meaning how much hunger they sate. One by one the feeding powers of the objects are compared to the need by absolute values, so that the result is always positive and thus easier to compare, until the current object is less suitable than the previous or the end of the list is reached. There were other faster, but less accurate ways of coming to a similar result, but in order to keep the AI's logic believable even in extreme situations (massive hunger, small foods; tiny hunger, big foods) this was the best solution.

When looking for a place to rest (sating energy), the AI currently primarily looks for objects that they "own", which in this case is their designated bed on the ship and if such no such object can't be found, they will simply go to the closest suitable object. This functionality of course is still very rudimentary, as the character will always go to their bed if they have one, even if they only want to have a short nap and are currently on the other side of the ship and near a suitable sofa. This functionality obviously has to be expanded upon, but as complex functionality wasn't within the scope of this thesis, these features will be left for future development.

For other needs the AI simply finds the closest object. Currently this is done by calculating the distance between the character and the object as a simple vector, but in the future it will most likely be replaced by a more robust system. As there might be many rooms and walls between the character and object, currently the AI will, under the right circumstances, choose the object with the shortest distance vector instead of the object within the shortest walking distance.

When the desired object has been chosen, the FSM changes to its final state (actionState == 2), at which point the control of the character is given to the smart object. As the FSM is not needed while the object is in charge, it goes idle (actionState == 3) and waits for outside input.

6.4 Moving characters around the ship

Before starting work on the AI's logic for using the objects, I decided to tackle the matter of navigation. At a very early point in designing the game, I entertained the idea of making the whole game text based with no visual components to circumvent the problem of navigation entirely. The idea was quickly scrapped, because the game had to have a time system and moving a character around the ship instantaneously wasn't an option and calculating movement times across different sized rooms would've only made things more difficult. Ultimately I came to the conclusion, that making a physical simulation of the ship and characters was the best solution.

Next I had to decide what method of navigation to use. At first I thought about using a waypoint system, because of the simplicity of the ship's design, but already during the design phase I quite quickly realized the limitations and problems with the approach. Every waypoint would have to be placed manually to guarantee correct placement, the connection between waypoints would have to be made by hand and every room of the spaceship would most likely require close to ten waypoints. Adding a new room or modifying an old one would also mean reworking the waypoints of multiple rooms. It's easy to see why this doesn't seem like a good solution. After that I went with my secondary idea, which I quickly realized was the best approach: using navmesh navigation.

I knew beforehand that Unity has built-in navmesh functionality – with which I was already somewhat familiar with - so building the actual navmesh was fast and easy. I only encountered one problem during the whole implementation process. At first, after building the physical mesh, the navmesh itself wouldn't bake. I spent a lot of time playing around with the navmesh baking options, suspecting they were the problem, but ultimately the problem was due to Unity's internal logic. Like all 3D programs, Unity has an x-y-z –coordinate system, which in Unity's case goes: x-axle is width, y-axle is height, and z-axle is depth. I had built the ship on the x-y –plane, meaning the ship's bow was pointing upwards and since Unity's physics engine uses the x-z –plane as "ground", it couldn't bake an "upright" navmesh. After I rotated the ship 90 degrees, so that it was on the x-z –plane, the navmesh baked perfectly.

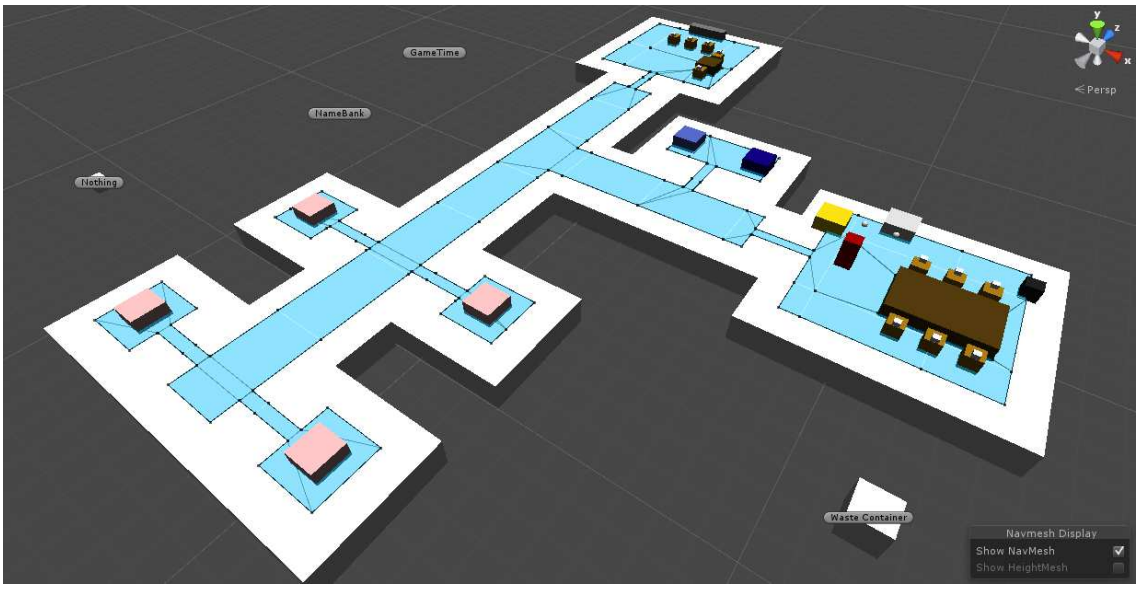

FIGURE 9. Screenshot of the ship and its navmesh in Unity (Unity Technologies 2014, viitattu 22.12.2014)

After the navmesh was complete, the characters must be given Unity's "Nav Mesh Agent" – component with which they can be moved around the ship. This requires only a single line of code and the "Nav Mesh Agent" takes care of the rest.

6.5 Implementing smart objects

In my eyes, when designing an AI, the aim is to boil its logic down to the simplest form you can. This makes the actual programming of the AI much easier, as its entire logic can be executed with a handful of methods. I've learned this the hard way in the past, when I designed a "perfect" Tic-tac-toe AI. The AI couldn't lose and would always aim to win, but as I was unable to boil down it's logic to a simple enough form, programming it became a nightmare. I would've had to program each individual situation and outcome separately, because I couldn't come up with functions that could cover every situation. To avoid the same pitfall with this project I thought very long and carefully about how the core logic of sating the numerous needs would work. All, or at least most of the objects would have to be able to function with a limited set of general functions, so that programming the AI wouldn't become a massive task.

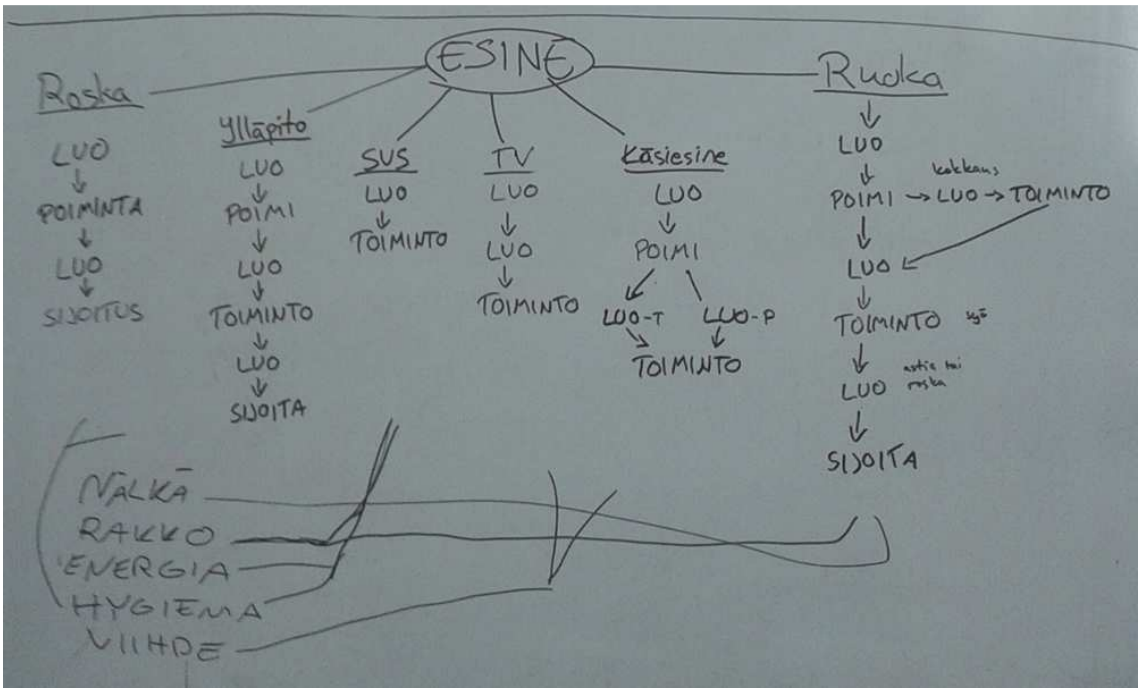

FIGURE 10. The object AI structure being designed

Since Spaceship Captain is a top-down 2D-game, the AI doesn't have almost any visual component like in The Sims, where the characters must have unique animations for all the objects they interact with. In this prototype-phase I only focused on the logic of the AI, which meant it was enough that the AI only moves to the designated object with no animations. This could have made it possible to program the entire logic of the AI into the characters themselves, but instead I decided to use "smart objects" like in The Sims. Though the actual use of the smart objects wouldn't be exactly the same as in The Sims, they would still provide some of the same advantages: more easily maintainable code, which in turn makes editing and adding new object less of a hassle. The script for the AI characters was already nearing 300 lines of code, which is already somewhat unwieldy, so adding the logic of all the objects in the game into the same script would've made it an absolute horror to debug.

Because of the work I had put into designing the object AI, I could translate it's logic into a handful of reusable functions for most objects and so utilize a programming method called inheritance. Inheritance works so that a base-class or "parent-class" contains all the variables and functions the sub-classes or "child-classes" inherit all the contents of the base-class, but can modify and add to them. Using inheritance saves me from rewriting code, but it also makes it more reusable and easier to follow.

After defining the necessary functions in the base-class, constructing the "structure" of the objects AI was straightforward. The core of the object AI is a simple FSM, controlled by a single variable, “state”. A secondary variable, “doOnce” is used in some cases to limit the execution of some functions to a single execution. Every state of the FSM is assigned a function from the base-class or in some cases written in the object itself.

```
8   public override void Use() {
9       if (state == 0 && doOnce) GoToObject(gameObject);
10      else if (state == 1 && doOnce) PickUpSelf();
11      else if (state == 2 && doOnce) GoToClosestObject("Chair");
12      else if (state == 3) ChangeNeed();
13      else if (state == 4) MakeTrash();
14      else if (state == 5) DestroySelf();
15      else if (state == 8) EndInteraction();
16  }
```

FIGURE 11. FSM of a food object in the prototype

Though the "structure" of the AI's logic is contained in the objects themselves, some of the AI's functions are actually executed by the characters themselves. For example, this is the case with the first function seen in figure 11, "GoToObject()", which is defined in the base-class.

```
32  public void GoToObject(GameObject obj) {
33      // != "Room"
34      if (obj.transform.parent != null) user.gameObject.
35          SendMessage("GoToObject", obj.transform.parent.gameObject);
36      else user.gameObject.SendMessage("GoToObject", obj);
37      doOnce = false;
38  }
```

FIGURE 12. "GoToObject()"-function in the base-class for all objects

In the "GoToObject()"-fuction, the object sends the "user" (character using the object) a command to execute another identically named function in the character AI and as a parameter gives the object itself or its parent object.

```
252  public void GoToObject(GameObject obj) {
253      destinationObj = obj;
254      agent.SetDestination(obj.transform.position);
255      print("Going to " + obj.name);
256  }
```

FIGURE 13. "GoToObject()"-function in the character AI

Finally the "GoToObject()"-function in the character AI gives a command to Unity's "Nav Mesh Agent" –component, which then starts moving the character towards the object given as the parameter (the food object).

By building the AI structure of the object like this, adding new objects and editing existing ones is very simple. The scripts attached to the objects stay short and easy to read as in most cases they only consist of the "Use()"-function, while the actual logic is kept in the base-class and the character AI.

7 RESULTS AND DISCUSSION

When I started working on this thesis, the biggest problem I had, was how to limit the scope of it. At first I thought the need-based functionality would be too limited of a subject, but as I started planning out the structure of the thesis and eventually writing the content, it turned out that the need-based functionality was more than enough to make the thesis out of.

The question of whether or not the goal of the thesis was achieved is still something depends on the point of view of the argument. The core of the AI's need-based functionality was successfully developed, but not all of the documented needs made their way into the prototype. They could've easily been implemented to some degree, but not in the way they would've functioned in the final game. The needs would have needed other supporting systems, which would've blown the scope out of proportion in this thesis and extended the development time of the prototype significantly. I would've liked to develop the prototype further and especially start working on the social abilities of the AI, as it's one of its key aspects for the finished game. Its complexity would've however made it more suitable as its own thesis as a continuation of this one. Also the fact that the design of the social aspect of the AI is still painfully unfinished meant that they had to be left out.

Overall I'm quite satisfied with this written part of the thesis. It's quite clear and cohesive work and in my own opinion manages to showcase a plethora of design and programming challenges in a practical light. However, I've been somewhat disappointed at the state I had to leave the actual prototype in. In order to finish this thesis on time and other time constraints, I had to stop the development of the prototype what I would say partway. Because of this, a few of the new features were only implemented partially, but then again, as they weren't a part of this thesis, this is just a personal gripe. However, there were some shortcomings in the need-based functionality I would like to cover in more detail.

The decision making algorithm is something I most likely still have to put more work into in the future. In current testing it does its job fine, as it cannot be tested in actual gameplay until the development of the game reaches alpha or beta stage, it might turn out to be unsatisfactory later on. It most likely can be improved with simple testing and tweaking the values, but during my research into The Sims in the latter phases of this thesis, I started considering using a similar system to the Happiness-system (chapter 5) instead of my current state-based system or a mix of

the two. However, as such a major change of mechanics requires a lot of further design and research before any concrete decisions can be made, I decided not to tackle it now and leave it for future development.

Another, closely related issue is interrupting actions. This functionality isn't yet implemented, but is high on the list of future features, because as the prototype is right now, the AI can't choose a new action until the current one is finished. Though not by far the worst problem, in extreme cases it makes the AI's behavior highly unbelievable. If for example a characters bladder and energy needs are both low and the character decides to sleep and during sleep his bladder would go all the way down, the character wouldn't wake up, but wet his bed every time. This quite obviously wouldn't be very believable behavior for rational adults. The most obvious solution would be to have a "critical point" for all the needs, which when reached could have a chance of interrupting the current action. Another solution would be to have multiple "critical points" or a dynamic interrupting system, that alerts the character if the current action's remaining execution time will drain a need completely.

There is also an issue regarding the smart objects in the prototype. Unlike in The Sims where the smart objects can advertise multiple needs, in Unity each game object can only have one tag, and as discussed earlier in chapter 6.3, the prototype currently uses tagging to advertise the needs the objects sate. This wasn't a problem for the needs of this thesis, but in the future this limitation has to be circumvented. I've seen multiple people request the ability to use more than one tag per object, but as Unity Technologies is yet to confirm the feature for future releases, the only solution I see is to write a custom system to handle the advertising functionality.

Another issue that wasn't really relevant to this thesis, but one that I realized while writing was the way AI checks its distance to objects during the decision making process. Currently the AI calculates the distance between the character and the object as a simple vector. In the future, the level (i.e. the spaceship) is most likely going to become a more complex structure, meaning there might be many rooms and walls between the character and object. Using the current system, the AI will ignore all walls and always choose the object with the shortest distance vector instead of the object within the shortest walking distance. Under the right circumstances this result in unrealistic behavior. A possible solution to this problem would be to generate a navmesh path to each object and compare their lengths instead of simple distance vectors, but I have yet to test whether this feature is present in Unity.

SOURCES

Abraham Maslow. 2009. Abraham Maslow Biography. Referenced 22.12.2014, http://www.abraham-maslow.com/m_motivation/Biography.asp.

AI Game Programmers Guild. 2011. The Sims. Referenced 16.12.2014, http://gameai.com/wiki/index.php?title=The_Sims.

Champandard, A. 2007. Living with The Sims' AI: 21 Tricks to Adopt for Your Game. Referenced 2.12.2014, http://aigamedev.com/open/review/the-sims-ai/.

Cherry, K. 2014. Biography of Abraham Maslow (1908-1970). Referenced 22.12.2014, http://psychology.about.com/od/profilesmz/p/abraham-maslow.htm.

Electronic Arts. 2010. Needs TS1.jpg. Referenced 26.11.2014, http://sims.wikia.com/wiki/File:Needs_TS1.jpg.

Epic Games. 2012. Navigation Mesh Reference. Referenced 2.12.2012, http://udn.epicgames.com/Three/NavigationMeshReference.html.

Factoryjoe. 2009. File:Maslow's Hierarchy of Needs.svg. Referenced 26.11.2014, http://commons.wikimedia.org/wiki/File:Maslow%27s_Hierarchy_of_Needs.svg.

Microsoft. 2014a. Application Development. Referenced 2.12.2014, http://www.visualstudio.com/explore/application-development-vs.

Microsoft. 2014b. Visual Studio with MSDN. Referenced 2.12.2014, http://www.visualstudio.com/products/visual-studio-with-msdn-overview-vs.

Mark, D. 2009. Behavioral Mathematics for Game AI. Boston: Cengage Learning.

PaulT. 2008. Fixing Pathfinding Once and For All. Referenced 2.12.2014, http://www.ai-blog.net/archives/000152.html.

McLeod, S. 2014. Maslow's Hierarchy of Needs. Referenced 22.12.2014, http://www.simplypsychology.org/maslow.html.

Unity Technologies. 2013. Unity Releases 2D Tools With 4.3 Update. Referenced 19.11.2014, http://unity3d.com/company/public-relations/news/unity-releases-2d-tools-43-update.

Unity Technologies. 2014a. The Leading Global Game Industry Software. Referenced 19.11.2014, http://unity3d.com/public-relations.

Unity Technologies. 2014b. Create the Games You Love With Unity. Referenced 19.11.2014, http://unity3d.com/unity.

Unity Technologies. 2014c. License Comparison. Referenced 19.11.2014, http://unity3d.com/unity/licenses.

Unity Technologies. 2014d. Creating and Using Scripts. Referenced 19.11.2014, http://docs.unity3d.com/Manual/CreatingAndUsingScripts.html.