Imitation of Player Behavior in Sandbox Games using Unity Machine Learning Agents Toolkit

Bachelor thesis by Thomas Haase (732830)
Summer Semester 2018

Hochschule Darmstadt
Digital Media – Animation and Game

Erstprüfer
Prof. Dr. Martin Leissler

Zweitprüfer
Thomas Nickel

Table of Contents

0. Abstract

0.1. Abstract – English

Repetitive tasks in Sandbox Games obstruct players to play cooperatively and to experience other, more interesting content of the game. This problem can be fixed with automation of these repetitive tasks by means of Artificial Intelligence (AI). This automation has to be accessible for any kind of player. That is why Imitation Learning is employed to imitate behavior desired by the player. The prototype of a Sandbox Game "Dominus" implements Unity's Machine Learning Agents to test the capabilities and limitations of this technology. The prototype shows learning abilities of the algorithm, its performance compared to conventional Utility-based AI and complications during the process of its implementation. The results appear for Imitation Learning to be the right choice in theory, but the current state of the technology and the implementation do not provide the means to solve the problem sufficiently. Nonetheless, the insights gained throughout the process present additional approaches, that promise to enhance the implementation to be a successful and viable attempt.

0.2. Abstract – German

Repetitive Aufgaben in Sandbox Spielen verhindern das kooperative Zusammenspiel der Mitspieler und zwingen sie viel Zeit in einzelne Teilbereiche des Spiels zu investieren, welche sie vom gewünschten Inhalt des Spiels abhalten. Automatisierung dieser repetitiven Aufgaben durch Künstliche Intelligenz (KI) könnte das Problem lösen. Dieses automatisierte System muss ohne weiteren Aufwand für jede Art von Spieler verwendbar sein. Deshalb soll Imitation Learning eingesetzt werden, um vom Spieler bevorzugtes Verhalten zu imitieren. Es wird ein Prototyp Sandbox Spiel „Dominus" geschaffen, der Unity's Machine Learning Agents Toolkit implementiert, um die Möglichkeiten und Einschränkungen dieser Technologie zu testen. Der Prototyp zeigt auf, welche Lernfähigkeiten der Algorithmus besitzt, welche Leistung er gegenüber herkömmlicher Utility-basierter KI erzielt und wo Probleme in der Implementation auftreten können. Es wird offensichtlich, dass Imitation Learning prinzipiell die richtige Wahl für das Lösen des Problems ist, aber auch dass die Technologie und die Möglichkeiten der Implementation in der im Prototyp vorhandenen Form nicht ausreichen, um das gewünschte Ergebnis zu erzielen. Aus den Erkenntnissen können allerdings weitere Schlussfolgerungen und Ansätze gezogen werden, die eine erfolgreiche Implementation von Imitation Learning in Aussicht stellen.

1. Repetitive tasks obstructing cooperative gameplay

From the year 2006 Minecraft [2006, Mohjang] dominated the Sandbox Survival genre and its sales reached 144 million across all platforms in 2018.[1] It probably hit a nerve since its very beginnings: players could follow their own idea of what the game should be about. Other than in story driven games or games with a dedicated goal, in Sandbox games there is generally no overarching primary goal to achieve. There typically is no end screen to the game, congratulating the player for beating the last level or final boss of the game. The Sandbox game goes on indefinitely.

The content of a Sandbox game can otherwise vary, for example physics-based simulation, economic simulations or Roleplaying Games (RPGs). The term Survival indicates the need for the player to do something in order to not lose. This often includes gathering resources and creating some form of safety. The primary source of fun to the player is that the way to go about this is open to her creativity and that the any goals to reach are set by herself.

In terms of Bartle's Taxonomy[2], originally found for MUDs (Multi User Dungeons), there are ways to play the game as achievers, explorers, socializers or killers: Achievers "*give themselves game-related goals, and vigorously set out to achieve them*". Explorers *"try to find out as much as they can about the virtual game world".* Socializers *"use the game's communicative facilities, and apply the role-playing that these engender, as a context in which to converse (and otherwise interact) with their fellow players".* Killers *"use the tools provided by the game to cause distress to (or, in rare circumstances, to help) other players".*

While this taxonomy is not exclusively endemic to cooperative Sandbox Survival games, it becomes obvious that the range of variety is greater here than in other genres, e.g. when compared to an Action Shooter game, like Counter-Strike, where exploration hits its limits with the map's confinements and the amount of weaponry.

Usually other Sandbox Survival games, like 7 Days to Die, Ark, Space Engineers or Don't Starve, share this kind of play styles. But they do also share the same problems:

The game invites players to play cooperatively, but the shape of cooperative play often disintegrates into several parallel tasks for individual players. Even though a greater goal is achieved together, the single tasks to fulfill this greater goal will often be taken on by a single player for each task. In terms of efficiency and coordination it absolutely makes sense to the

[1] "Minecraft Sales Reach 144 Million Across all Platforms; 74 Million Monthly Players" *Wwcftech*, 1. Jan. 2018. https://wccftech.com/minecraft-sales-144-million/ Accessed Aug. 2018

[2] Bartle, Richard. "Hearts, Clubs, Diamonds, Spades: Players who suit MUD" Apr. 1996. http://mud.co.uk/richard/hcds.htm Accessed Aug. 2018

player to have specialized roles, but it also diminishes the intention of cooperative play in short-term. When a group of players decide to build a home-base, they might need defenses, food and resources, as well as some items only acquired outside the home-base. So, while one player sets out to the world to find rare items, another is tasked to build walls and traps, someone else raises and maintains a farm and yet another grabs a shovel to dig in the ground for precious minerals. Some of these tasks, like mining, can be very repetitive and go on indefinitely, while others require arrangement and coordination of players to not obstruct each other's endeavor, like building a base. Therefore, it is more effective to the players to distribute tasks among themselves. Depending on the game, they might even be forced to do so, when enemies menace the base in the short-term and weather conditions threaten the group's food supplies in the long run.

For that reason, the freedom for players to choose their own play-style succumbs to the demands of the group and the requirements for the group to succeed in the game environment. Of course, this can be a deliberate choice for the game's design: A shared greater goal[3] and the successful reaching of it produces a sense of group achievement. Specialized roles create a sense of identity and ownership of a player in the group. Also, repetitive grinding is a valid design choice to increase the game's longevity and to induce cognitive flow[4] in the player.

This works well in games like World of Warcraft [2004, Blizzard] or Grand Theft Auto Online [2013, Rockstar North], when players are close by, separated only for short times and work together on short-term goals. They engage the common goal in direct relation to each other. But in Sandbox games, players can be far from each other for a long time. The cooperative aspects only appear in the long term, as the short-term tasks need to be done. The common goal is only achieved indirectly with other players in the group.

It is interesting to look at common Sandbox design choices and single out the components, that support cooperative gameplay in a way that players can engage directly with each other on a more regular basis. To enhance direct engagement, it could suffice to reduce the necessity for constant, active pursuit of repetitive, indirectly connected tasks, without

[3] Staats, David. "Designing Cooperative Gameplay Experiences" *Gamasutra*, 15. Dec. 2015. https://www.gamasutra.com/blogs/DavidStaats/20151221/261927/Designing_Cooperative_Gameplay_Experiences.php Accessed Aug. 2018

[4] Baron, Sean. "Cognitive Flow: The Psychology of Great Game Design" *Gamasutra*, 22. Mar. 2012 https://www.gamasutra.com/view/feature/166972/cognitive_flow_the_psychology_of_.php Accessed Aug. 2018

removing them from the game entirely. They belong to the players' common knowledge of what is expected of a Sandbox Survival game. Should a player choose to grind for minerals, he still should have the option to do so.

So, how can the time spent playing alone in a cooperative game be reduced to crisp, meaningful sections to enhance direct cooperative efforts?

2. Automation and Artificial Intelligence

In the history of games, that include amassing some kind of material gain, there always were lazy (or smart) players, that decided they have better things to do with their time than repeatedly doing the same thing over and over again but did not want to miss out on the benefits in doing so. So, they employed third-party tools, namely bots, to do the chores in a simplistic fashion. These might be simple fishing bots for World of Warcraft, or even literal robotics, that push the keyboard's space bar every second to increase the characters Jumping skill in The Elder Scrolls III: Morrowind [2002, Bethesda Softworks]. This even goes so far as bots in competitive games, either destroying the opposition with super-human capabilities or performing subpar to collect consolation prizes at the end of a doomed match.

When these bots do not make the game pointless to other players in Counter-Strike [2000, Valve] or League of Legends [2009, Riot], they at least give those crafty players an advantage over their peers. No wonder that game developers try to counteract by using their own tools to combat this form of cheating. Usually this means a ban of the player account from the game, at first for a reasonable amount of time but possibly permanently.

But the craftiness of those cheating players merits some attention, perhaps even admiration. The evolution of technical advance of mankind was always about making life easier. Of course, this does not excuse the spoiling of other players' gaming time or the circumvention of deliberate design choices. Even though, some games reward smart players that use automation. In those cases, no third-party tool is needed, but the tools are already built in, for example the Redstone-circuitry of Minecraft, or the conveyor belts and inserters of Factorio [2016, Wube Software].

Typically, this kind of enjoyment is rather niche and it takes effort to set up such automation processes to the point where players decide to dismiss this option entirely. To open up automation for everybody to enjoy, it would need a system that does not require a lot of learning or complicated logic for players to use it. Ideally, the player should not need anything new to learn and, instead, show the system what he wants it to do by example.

This is where the technology of Imitation Learning comes into play, as it promises to do exactly that. A presentation of Unity's Machine Learning Agents at the convention Unite Berlin 2018[5] claimed that it "Doesn't take long to mimic behavior" and showed an example for the racing game Antigraviator [2018, Cybernetic Walrus]. In the example one player-controlled racer provided the machine learning agent, who was controlling another racer, with training data. After only 30 seconds of training, the AI controlled racer is starting to turn in the right moments. After five minutes of training it is hard for an observer to distinguish between the behaviors of the player-controlled racer and the agent-controlled racer.

*Figure 1 – Antigraviator**

Should Unity's Machine Learning Agents Toolkit (MLAgents) perform similarly well in more complex scenarios, then it is adequate for using it for automation of repetitive processes in Sandbox Games. A prototype of a Sandbox Game, that includes the most common game mechanics of the genre, will test the toolkit's performance. The prototype's code architecture has to support the requirements of Sandbox Games and for the MLAgents Toolkit. The prototype has to be able to show the learning capabilities of the algorithm to make a decision possible, whether this technology provides the results needed to employ it in a commercial environment in more complex scenarios. Or in other words, is it feasible to use MLAgents Toolkit's Imitation Learning algorithm by Unity in a way, that allows players to train characters to do repetitive work in a Sandbox Game easily and reliably?

[5] Berges, Vincent-Pierre"Democratize Machine Learning" *Unite Berlin*, 21. Jun. 2018. https://www.youtube.com/watch?v=a768FLX9bRc&t=2h30m Accessed Aug. 2018

3. State of the Art

3.1. Game Engines

Game Engines provide a fast and easy way to create games and prototypes for professionals and hobbyists alike. The availability of a game loop, a render engine etc. makes it possible to focus on the game itself. With some products, like the Unreal Engine 4 [2014, Epic Games], not a single line of code is required to create a functioning game. Other renowned game engines are Source Engine [2004, Valve] and CryEngine V [2016, Crytek]. Most of these engines are free to use and experiment with up to a commercial release of a game, when their developers take royalties of the profits.

For the purpose of the prototype for Dominus the game engine Unity3D [2004, Unity Technologies] had to be employed, as the technology used to provide Imitation Learning, namely Unity Machine Learning Agents Toolkit, was created specifically for this game engine.

3.2. Machine Learning Software

TechEmergence, the "*[...] industry source for authoritative market research and competitive intelligence for [...] artificial intelligence*"[6] defines Machine Learning like this:

"Machine Learning is the science of getting computers to learn and act like humans do, and improve their learning over time in autonomous fashion, by feeding them data and information in the form of observations and real-world interactions."[7]

Machine Learning software is used in business analytics, finance, healthcare, marketing, robotics and security. Some examples for open-source software are Deeplearning4j[8], OpenNN[9] and TensorFlow[10]. The proprietary software includes big names like Amazon Web Services[11], Microsoft Azure[12] and Oracle Data Mining[13]. This shows, that machine learning is commonplace in today's business world. In games there are less examples.

Unity's Machine Learning Agents Toolkit is an attempt to democratize machine learning, so more game developers can experiment with the technology, just like it is done in this Bachelor thesis. It serves as middleware between Python-based TensorFlow and the Unity3D game engine. Unity's Machine Learning Agent will be used in the prototype "Dominus".

---

[6] "About TechEmergence" *TechEmergence.* https://www.techemergence.com/about/ Accessed Aug. 2018
[7] Faggella, Daniel. "What is Machine Learning?" *TechEmergence*, 2. Sep. 2017 https://www.techemergence.com/what-is-machine-learning/ Accessed Aug. 2018
[8] *Deeplearning4j*, 2017. https://deeplearning4j.org/ Accessed Aug. 2018
[9] *OpenNN Neural Networks*, 2018. http://www.opennn.net/ Accessed Aug. 2018
[10] *TensorFlow.* https://www.tensorflow.org/ Accessed Aug. 2018
[11] *Amazon Web Services*, 2018. https://aws.amazon.com/ Accessed Aug. 2018
[12] *Microsoft Azure*, 2018. https://azure.microsoft.com/ Accessed Aug. 2018
[13] "Oracle Data Mining" *Oracle.* http://www.oracle.com/technetwork/database/options/advanced-analytics/odm/overview/index.html Accessed Aug. 2018

3.3. Example Sandbox Games

3.3.1. Minecraft as paragon of Sandbox crafting games

The most famous example of a Sandbox crafting game probably is Minecraft, released in May 2009 by Markus "Notch" Persson. To this day it is continuously being developed and was recently released for the Nintendo Switch[14]. It features procedurally 3-dimensional generated terrain and biomes, crafting, equipment, interactable objects, hunger, pets, monsters, towns of Non-Player Characters (NPCs) and multiplayer. It is easy to create modifications (mods) and a lot of unofficial content enriches the games of enthusiastic players to this day. One of these mods, Minecraft Hunger Games, is even considered the predecessor of the whole genre of Battle Royale[15]. Sandbox games in particular often are measured in respect to the success of Minecraft. Some of these Sandbox games following in the wake of the Sandbox hype are Don't Starve [2013, Klei Entertainment] , 7 Days to Die [2013, The Fun Pimps], Terraria [2011, Re-Logic], Space Engineers [2013, Keen Software] and Factorio. Each of these titles emphasized on a certain aspect of their paragon, Minecraft, but still kept the core formula.

3.3.2. Sandbox games based on Minecraft

While Don't Starve focuses on the hunger mechanic, 7 Days to Die tries to enhance the danger to the player imposed by zombies. Both games thereby create a sense of common enemy to encourage players to cooperate. Terraria takes the concept to a 2-dimensional world and introduces boss encounters and equipment of varying quality to give it more of an adventure style touch. Here, too, the common short-term goal of defeating the enemy invites players to join forces. All of the above attempts, including Minecraft, rely on repetitive tasks done by solitary players over elongated times for progression of the group.

3.3.3. Sandbox games with Automation

Space Engineers takes the concept to zero-gravity with physics-simulated destruction and introduces the possibility to program behavior of constructions with LUA scripts within the game. Automation makes up most part of Factorio, where conveyer belts and automated workshops continuously produce resources, while the player constantly expands the process. In both cases attempts were made to reduce repetitive tasks with automation. But to do so players have to either, in case of Space Engineers, have skills in programming, or have to split responsibilities to not obstruct careful planning of efficient building of a factory in Factorio.

---

[14] "Minecraft: Nintendo Switch Edition" *Nintendo*, 12. May 2017. https://www.nintendo.de/Spiele/Nintendo-Switch/Minecraft-Nintendo-Switch-Edition-1214741.html Accessed Aug. 2018

[15] Green, Ollie. "Minecraft: Hunger Games is the best Battle Royale Game" *GameByte.* 10. May 2018 http://www.debate.org/opinions/minecraft-hunger-games-or-fortnite-battle-royale Accessed Aug. 2018

3.3.4. Sandbox games with Artificial Intelligence

Not all Sandbox games derive their formula from Minecraft. In fact, one of the influences for Minecraft was Dwarf Fortress [2006, Tarn Adams], where the player manages a kingdom of up to 200 dwarves, which all have their own desires, properties, characteristics and quirks. The dwarves act autonomously based on their traits and current needs, similar to another famous Sandbox game, The Sims [2000, Maxis]. Both of these games employ Utility-based AI to control NPCs. This has the advantage of the player not being needed at all times for each character. The player has the time to experience the content, that is most interesting to him. Both of these games are not meant to be played cooperatively with other players, though. While the problem of repetitive tasks obstructing cooperative gameplay does not appear in this context, it is still an interesting solution to use AI when the player wants to take her attention to something else.

3.4. Example Machine Learning Games

Black & White [2001, Lionhead Studios] is not a Sandbox game, as there is a clear goal to the game. The player has to impress NPCs with miraculous presentation of her godly powers to grow her radius of influence and then attack her antagonist's base until it breaks. The player is given a mostly autonomously acting creature, which makes it possible for her to interact with the game world out of his radius of influence. She can also teach the creature to perform tasks like fetching food or lumber, performing miracles or attacking enemy towns. When the creature does the right thing, the player can reward it. When it does the wrong thing, the creature could be punished. The ideal taught creature would be able to play the game without the need of the player. The neural network used here resembles Reinforcement Learning and comes close to what the solution of the problem of repetitive tasks in Sandbox games could be about.

Creatures [1996, Cyberlife Technology] is an attempt by Steve Grand to create artificial life. In his pursuit he created the game as a byproduct. The characters in the game, called Norns, are childlike autonomous creatures, controlled by a neural network. Norns have virtual DNA, and a biochemical system, which means they have needs for certain types of nourishment, can get sick and also reproduce. The player has no direct control over the Norns, but can influence the game world and can reward and punish the Norns for good and bad behavior, similar to Black & White. Both examples show the possibilities for Machine Learning to create lifelike behavior in NPCs.

4. Related Work

Unity's Machine Learning Agent Toolkit is a fairly new product, as it was only introduced in September 2017.[16] While Machine Learning has some use-cases in games, it is not commonplace technology. This is reasonable when considering the unpredictability of the learning process. The lack of authorial control makes it hard to debug errors both in the environment and the setup of the algorithm. TensorFlow is working hard on visualizing the setup of machine learning algorithms, which makes debugging easier. The impact of TensorGraph[17] on the games industry remains to be seen. Nonetheless, conventional AI is more reliable in terms of debugging and production time to large game companies, that cannot afford risks in productions with massive budget. For that reason, it comes down to independent developers, hobbyists and enthusiasts to explore possibilities, limitations and pitfalls of the technology.

Unity's Technical Evangelists are constantly exploring ways to convey Machine Learning potential pioneer developers, while technicians react to requests and bugs fixes suggested by the active developer community. The best resources at this moment to get a first glance at Unity's Machine Learning Toolkit are the blog posts "*Using Machine Learning Agents Toolkit in a real game: a beginner's guide*"[18] and "*Imitation Learning in Unity: The Workflow*"[19] by Alessia Nigretti.

For a deeper understanding of the implementation of the Toolkit in a project is the Git-repository[20] and the documentation therein. At this point, there is no citable paper for the technology, but the creators have announced some document to come.[21]

The first competition hosted by Unity Technologies "ML-Agents Challange 1"[22] from January 2018 featured 43 submissions. None of the challengers tried to get results similar to what is proposed in this paper.

---

[16] Juliani, Arthur. "Introducing: Unity Machine Learning Agents Toolkit" *Unity Blog*, 19. Sep. 2017. https://blogs.unity3d.com/2017/09/19/introducing-unity-machine-learning-agents/ Accessed Aug. 2018

[17] "TensorBoard: Graph Visualization" *TensorFlow,* 19. Jul 2018. https://www.tensorflow.org/guide/graph_viz Accessed Aug. 2018

[18] Nigretti, Alessia. "Using Machine Learning Agents Toolkit in a real game: a beginner's guide" *Unity Blog*, 11. Dec. 2017. https://blogs.unity3d.com/2017/12/11/using-machine-learning-agents-in-a-real-game-a-beginners-guide/ Accessed Aug. 2018

[19] Nigretti, Alessia. "Imitation Learning in Unity: The Workflow" *Unity Blog,* 24. May 2018. https://blogs.unity3d.com/2018/05/24/imitation-learning-in-unity-the-workflow/ Accessed Aug. 2018

[20] Unity Machine Learning Agents Toolkit. *Unity Technologies*, 2018. https://github.com/Unity-Technologies/ml-agents

[21] "Issue #962: Citation of the project." *Unity Technologies*, 10. Jul. 2018. https://github.com/Unity-Technologies/ml-agents/issues/962 Accessed Aug. 2018

[22] "ML-Agents Challenge 1" *Unity Technologies*, 31. Jan. 2018 https://connect.unity.com/challenges/ml-agents-1 Accessed Aug. 2018

5. Imitation of Player Behavior using Machine Learning Agents

5.1. The Game Vision

Often gamers and game developers, professionals and hobbyists alike, fantasies about the perfect game. This leads to an overblown concept with a multitude of mechanics in the game, that might not be as much fun as expected and also might not be doable in a reasonable time frame. But it is fun to put everything into the basic concept and then reduce it from there. The result will be a game that is fun to play as well as doable by developers of a certain team size in a reasonable time frame.

5.1.1 Dominus - Game Design

The prototype “Dominus” is a top-down Sandbox Game, in which the player has the resources to plan and execute her own projects by means of artificially intelligent non-player characters (NPC). Individual NPCs act on their own to fulfill a multitude of needs by using skills of varying quality. The player can take control over an NPC for any length of time, which is called dominating. While being dominated NPCs record the behavior of the player and imitate this behavior when they are released from domination. NPCs are organized in towns with three to 30 inhabitants. Multiple towns can act aggressively or cooperatively towards each other.

5.1.1.1 Automated Content creation

The beauty about pen and paper roleplaying games (RPG), like Dungeons & Dragons [2003, Wizards of the Coast], is that you play it with the most flexible computer known to mankind, which is the human brain of the Game Master. When in a classical video game there is no content available behind a door, the door is usually locked. But the human mind can, based on the circumstances, spontaneously envision what should be behind the door and the Game Master can create that content as the players break down the door. Sure, some procedural generated content can do a similar thing, based on some preprogrammed rules. In a way this is Artificial Intelligence generating content. Modern machine learning algorithms could just as well be utilized to simulate a human mind for this, when done right.

5.1.1.3 Attributes and Skills

Pen-and-paper RPGs also give other insights, that help understanding what Dominus should be about. To simulate different characters, usually character sheets contain a multitude of statistics about the character’s abilities and conditions. A librarian usually has high Intelligence values, while a warrior depends on his Strength. A short glance on the character sheet tells the observer what kind of character is written there. He can also assume how the character would behave in different situations. While in a combat situation the warrior would thrive in the thick of bloodshed, a librarian would probably take cover or panic. On the other hand, a

simple puzzle could cause the warrior with low Intelligence quite a headache, while the librarian goes at the task with glee. This shows that with only a few statistics there already can be made distinctions between characters, which creates variety and ownership of the player for some situations. For that reason, characters in Dominus have Attributes and Skills determining the quality of possible success of actions within the game world. The pen-and-paper RPG Savage Worlds[23] provided statistics and values for an initial setup of Dominus.

5.1.1.4 Character Behaviors and Domination

How the characters behave, though, is still just an assumption of the observer or player. It is based on references from other media, like books, movies or video games. On the rational side, it is simply logical to have a character with high Vitality and Strength values in the frontline to soak damage, that would kill more fragile characters. But emotionally there can even be differences between two warrior types, like a cruel bandit king and a morally compassed paladin. Either way, the Artificial Intelligence of non-player characters in video games is limited to what the developer can accomplish. It simply cannot hold all the references about warriors on its own and recreate sensible behavior from those books and movies. These behaviors have to be created by the developer beforehand. But they could be instilled by those that already have the references, which are the players, instead. The advantage here is that the range of behaviors is not limited by the time frame of development and can grow as the game is being played. Imitation Learning seems to make this endeavor possible. By imitating the behavior of the player, the artificially intelligent non-player character can reproduce what the player imagines to be a sensible behavior for a certain character.

In Dominus the player would not control an actual character, that is able to interact with the game world. Instead she controls the Dominator, a parasite, that has the ability to take over control of any character for short and long periods of time, which is called dominating. While the player dominates a character, all actions performed by the player will be recorded by the Imitation Learning process. As a result, the dominated character will reproduce the behavior it was shown, as soon as it is released from domination. This enables players to specialize characters of his town to specific tasks and automate repetitive processes without the requirement for doing something else than actually playing the game. There is no need for the player to learn any logic or programming. The behavior of characters is steadily changed and

[23] "Savage Worlds - Gentleman's Edition Revised - Probefahrt" *Prometheus Games,* 2013. https://www.prometheusgames.de/download/savage-worlds/SW-GER-Probefahrt-v1.11-web.pdf Accessed Aug. 2018

improved well beyond the cycle of development and release of the game. The characters will behave in a fashion, which players assume to be sensible, provided they teach the Imitation Learner with sensible examples.

5.1.1.5 Motivation and Happiness

But there is more to a character's behavior than reacting to specific situations. People have a day-to-day life. In pen and paper RPGs these parts are often omitted due to two reasons. First, the attention of the players is limited to what they are being told by the Game Master, which in turn is limited by language. The Game Master is tasked with filtering the important information from the trivial. This leads to second: It is simply not as much fun to listen to detailed descriptions of every single character's comings and goings. In practice, a broad depiction of the situation will suffice until a specific information is important to what the Game Master has planned or what a player requests to know. Other information is left for the player's imagination to fill in the blanks. In video games, however, all of this information is displayed to the player when the game camera is angled at a situation. The player decides to what he will turn his attention to. In most games these situations are brief and most of the day-to-day life of an NPC is unimportant to the player. When the player enters a workshop in an action RPG, he expects the smith to do some crafting and be readily available to sell him weaponry or armor. Should the smith nervously rummage through some inventory, exclaiming frustration, this indicates to the player that there might be something interesting, perhaps a quest, at hand. This can even be done more simply with an NPC idly standing about and a big, yellow exclamation point hovering over his head. This might not be very lifelike, but usually players are forgiving as they, again, fill in the blanks with their imagination.

In simulation games, or god games, the NPC's behavior is visible most of the time. In any sandbox game the illusion of life, and thereby the immersion, can fall apart when the player decides to follow an NPC for an elongated time only to see that the NPC is running around in circles without a goal or reason. To give any NPC a motivation to pursue some reasonable goal at all times, characters in Dominus have needs to be fulfilled by several actions in the game world, similar to The Sims. Maslow's hierarchy of needs[24] acts as role model. Every character in the game is subject to its Happiness, a value determined by the number of sufficiently fulfilled needs. Happiness gives benefits, like faster movement, interaction speed and additional crafting recipes.

[24] McLeod, Saul. "Maslow's Hierarchy of Needs" *SimplyPsychology*, 2018. https://www.simplypsychology.org/maslow.html Accessed Aug. 2018

*Figure 2 - Maslow's Hierarchy of Needs**

It is easy to imagine an implementation of physiological needs, as many games already do. Minecraft and 7 Days to Die both use a hunger mechanic to force players to act. If they do not act to combat their character's hunger, they get punished by weaker performance of their characters or damage over time concluding in virtual death. Often two negative results can be observed: First, food supplies can run low in the most inconvenient moments of the game, which forces the player to interrupt his current endeavor to refill his supplies. This can happen at faraway places from stockpiles of food, so players have to move great distances for a long time, which keeps them from the actual fun content of the game. Second, the abundance of food available often comes down to two possible states. Either there is not enough of it, forcing the player to actively seek out sources, which can be the core aspect of a game like Don't Starve. In the other extreme, acquiring food is no problem anymore whatsoever as a result of smart planning and construction of farms by the player. In that case the hunger mechanic devolves to a frustrating experience of repetition and probably should not be in the game anymore. Both of these observable situations keep the players from experiencing other content, because of the extreme negative result in case they choose to ignore this hunger mechanic. The need "hunger" in this example is, of course, interchangeable with any other need provided by Maslow's hierarchy. For the sake of readability, all following examples will use this metaphor with resemblance to that assumption.

As characters in Dominus should actively pursue goals at all times, only one or two needs would not suffice to create interesting, diverse behavior. The fulfillment of needs is entirely optional, with the only drawback of missing out on considerable beneficial effects of high Happiness values. The Happiness missed out in one need can be compensated with Happiness from another. Needs from higher up the hierarchy produce more Happiness, but the maximum amount gained from high-up needs is capped by a multiplication of current amounts of Happiness in lower tiers. That way the hierarchical structure of the original role model is kept and early-game content is not rendered entirely pointless in later game progression.

5.1.1.6 Quality of Satisfaction

In other games a simple bar shows the amount of hunger. Different sources of food fill up the bar to varying satisfaction, depending on the quality of the food. In this case, three apples, which satisfy hunger by 5, are equivalent to a fruit-cake, which satisfies hunger by 15.

Should the acquisition of apples be a lot easier than baking a fruit-cake, the existence of the fruit-cake is rather pointless and the repetition in farming tons of apples increases as a result. To avoid this pitfall, increased quality of food in Dominus provides increased amounts of Happiness. Sources of food are not additive, so three apples do not add up to a fruit-cake, but one apple. Baking a fruit-cake, which provides thrice the Happiness, is therefore preferable. Needs, that are lower in Maslow's hierarchy, scale well into later progression of the game and do not become obsolete. Also, this removes repetition in the assumption of "quantity over quality".

5.1.1.7 Scalable Content and Interactions

To satisfy the characters needs and increase its happiness, the player (and the AI) needs to interact with the game world through an agent. The agent has functionality, that allows the player to do so. Exploration, as coined in Bartle's Taxonomy, can play a big part Sandbox Games, when players get the most satisfaction and fun out of the possibilities and content provided by the game. It is sensible for developers to provide fresh content on a regular basis to keep the game novel and intriguing. At this point, it is important to plan out the structure of the game environment, depending on the games content in terms of breadth and depth. It is not uncommon for naive developers and players to mistake the game's breadth for its depth. While a broad variety of food, like an apple, a banana and a pear, can seem to be a lot of content, it is essentially the same basic concept. Depth, in turn, is variety in interactions that can be done. In object-oriented programming this is a paragon for inheritance. But there

are problems at hand, when considering one behavior that one sub-class of Fruit should be able to do, a second behavior that a second fruit can do, while a third can do both. An example: An Apple can be planted to grow a tree, as can the Pear. An Apple can be juggled, so can the Banana (and for some mystical reason the Pear can't be juggled). So, the class Apple would need to inherit both behaviors from class Pear and class Banana. Next to the unintuitive reasoning of an apple being basically the same thing as a pear, inheritance is not capable to accomplish this. Usage of an interface demanding the implementation of a Juggle()-function in both the Banana and the Apple would lead to duplicated code. Each additional type of food, meaning each expansion of the game's content in terms of breadth, demands additional coding.

The Command programming pattern[25] provides a solution, that scales way better for additional content, but requires some additional work beforehand. The theory behind the pattern the and actual implementation in Dominus will be explained in detail in chapter 5.3.4. For now, it is important to note that in Dominus characters do not have any logic exclusively to their own, except movement. Instead, objects in the game environment hold a list of Interactions, that contain the functionality the original object should be capable of. When applied to the example above, an Apple would hold two Interaction-objects, PlantTree and Juggle, while the Banana would only hold the Interaction Juggle and the Pear would only hold the Interaction PlantTree. Should some developer decide for the Pear to be juggled as well, only the Interaction Juggle would be needed to be placed in the list of possible Interactions held by the Pear. The same way, a tree, a refrigerator or car could be juggled as well, without the need for extra code.

This comes back to the theory of depth and breadth: Filling the game with content in terms of breadth is especially simple, as functionality can be easily assigned to new objects by usage of already existing Interactions. Interactions can be seen as a measure for depth, as more functionalities mean more different things to do, which means more depth. Assuming a business environment for a commercial production of Dominus, this is highly advantageous when considering that in terms of breadth, no programmer is needed. Not only can dedicated content creators fill in a variety of objects, but also a possible fanbase can influence the game by adding or modifying content, e.g. via the Steam Workshop. Moreover, programmers have the time to work on the game's depth.

[25] Nystrom, Robert. "Game Programming Patterns" 2014.
http://gameprogrammingpatterns.com/command.html Accessed Aug. 2018

The decision to employ the Command pattern, with objects in the game world generating the Interactions, has impact on the AI as well.

5.1.1.8 Additional Features

Next to the core features already mentioned, there are mechanics in Sandbox games, that are typical for the genre and that Dominus should include at some point to make it a full, viable product. These include an inventory, so characters can collect and carry items for later usage. Also typical is crafting-system to combine items to give them greater value. Equipment is similarly common, which are items like clothing or tools, that enhance a character's performance and thereby create a sense of progression.

Other mechanics are needed for Domination to be viable. With just a single character to teach there would be no point in letting him lose on his own, while the player just watches. The creation and maintenance of an ecosystem, in form of a small town ranging from three to 30 inhabitants, gives the player ample opportunities to assign various characters to specific tasks. Opposing towns, controlled entirely by AI, give the player a measurement of his own performance and opportunities to interact with them aggressively or cooperatively, in forms of war or trade.

To give an edge to the need-mechanic and the aforementioned townsfolk, adjectives can change the importance of certain needs specifically to an individual character. An example would be an ascetic character, which halves the Happiness from satisfaction by food sources, but doubles the time until the satisfaction diminishes, or a gourmet being the opposite. Also, attributes like strength, agility or intelligence can further diversify the behavior of the characters, as already mentioned in the warrior and librarian examples. Social interaction between characters with a long-term effect on their behavior towards each other would round the experience of the game to a really lifelike one.

While all these mechanics are well beyond the scope of this thesis, it is interesting to keep them in mind when creating the code structure of the game and underlying architecture of the AI. Should the prove of concept for Imitation Learning in the prototype of Dominus prevail, these mechanics would be the next steps to take.

5.1.2 Content of the prototype

A project with a magnitude of what is suggested above can take years to be completed to a degree, that would satisfy the demands of a commercial market. A much simpler prototype will suffice to show the capabilities and limitations of Imitation Learning. As Occam's Razor states, things should not be made more complicated as they need to be. For Dominus this

means that only two needs, hunger and sleep, will be enough to show the concept of needs and Happiness. Also, the depth and breadth of items is reduced to only two items, apples and logs of wood, to show resources that either fulfill a need or can be utilized as building and crafting material. They can be picked up and carried in a character's inventory. Two characters are enough to improve and compare performances of the AI but will not allow for any social interaction. Characters will act in a confined space on a grid. The grid allows a pathfinding algorithm to efficiently work. The concept of skills can be proven by a single skill, WoodCut, which is needed to chop down trees. Two structures, trees and houses, are enough to provide the resources and to allow the fulfillment of sleepiness need. The prototype has implementations of two architectures of artificial intelligence, Imitation Learning and Utility-based AI. The former is needed to provide the learning capabilities, while the latter will be used as a teacher for the Imitation Learner and as an opportunity to compare performances.

5.2. The Paper Prototype

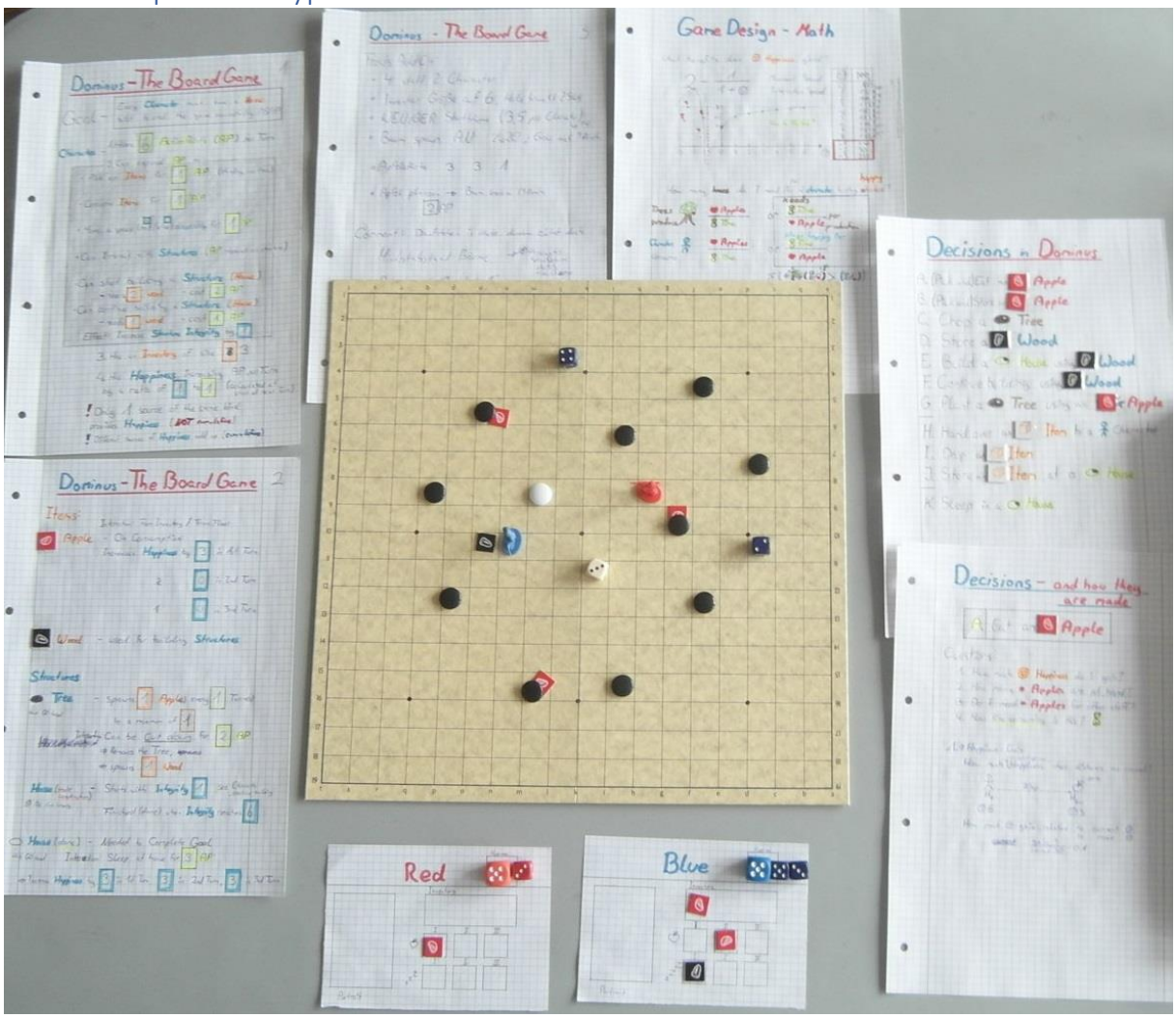

Figure 3 - The Paper Prototype

Creating a paper prototype helps a lot to come to conclusions about promising designs early in development. Lining out the rules so other players can understand is beneficial for clarifying all circumstances of the game. Should a player be confused about the rules, it is a good indicator for some mechanic not being correctly understood. When some rules are not used, then the design might just be not that interesting, in terms of fun and necessity. The goal for players in this paper prototype is for players to build one house for each character. There can be up to four characters participating.

Action Points (AP) indicate what time an action takes to perform. This makes it possible to map time scales of a real-time game to a turn-based board game. They are marked with dice on character's sheet. Each character starts out with six AP. This number can be increased by satisfying one or both of the needs hunger and sleepiness. When an action is performed to satisfy a need, it stays satisfied for the next three rounds. As more AP means more actions to be performed each round, it shows the speed of which the character moves through the game world. The amount of AP is calculated at the beginning of each turn:

6 + Happiness from satisfied hunger (0-3) + Happiness from satisfied sleep (0-3) = 6-12 AP

A character can perform the following actions:

- Move a character by one space (1 AP) – no diagonal movement
- Consume an apple (1 AP)
- Sleep in a house (3 AP)
- Pick up an item (apple or log of wood) (1 AP)
- Drop an item (1 AP)
- Pass on an item (1 AP)
- Start building a house (2 AP, 2 logs of wood)
- Improve a building (1 AP, 1 log of wood)
- Plant a tree (1 AP, 1 apple)

A character needs to stand next to the field where an action is to be performed on. A house needs a collective of 7 AP and 7 logs of wood invested, before a character can sleep there. A tree needs 3 rounds before it can grow an apple. When there is no apple on a tree at the beginning of a round, the tree starts growing an apple that will be available to characters at the beginning of the subsequent round. A character can carry a total weight of 6, while apples weigh 1 and logs of wood weigh 2.

The paper prototype showed that the content outlined with Occam's Razor was enough to pose interesting problems for players to solve. Most of these problems revolve around calculating of movement and distances to increase efficiency, planning placement and number of trees and houses and managing the limited inventory, while maximizing the AP for following turns. For example, carrying 3 logs of wood can have the risk of losing the benefits from satisfied hunger, but the construction of the first house can be completed after this one trip. Carrying only 2 logs of wood, but also taking 2 apples as well, guaranties the safety of 3 additional points, but the satisfaction of sleep will have to wait for an additional trip to the nearest trees and back. The more efficient solution depends heavily on the placement of trees and the construction site. The paper prototype made it possible to predict suitable values for various data in the game as well, e.g. the number of trees per character at the start of the game.

Several playtests indicated that a similar setup of the actual implementation of Dominus in Unity would provide interesting problems to an artificial intelligence. Although the imitation of player behavior is paramount in the Dominus prototype, the design of the Utility-based AI can benefit from observation during playtesting the paper prototype as well.

5.3. The Underlying System

Careful consideration was important prior to programming a vertical slice of Dominus in form of a playable prototype. As programming AI can be confusing and, at some points, hard to debug for odd behaviors, it is paramount to have clean game code. It was not uncommon for tests to be rendered utterly useless, when a bug in the game code allowed actions that the AI would exploit mercilessly. On the other hand, some bugs made the Machine Learning agent completely blind in a training session that was running for over six hours, the error being a division of integers (8 / 16 = 0) instead of a division of floats (8f / 16f = 0.5f). Therefore, a sustainable architecture was necessary to keep the code clean at all times and to provide opportunity for an ongoing production of Dominus beyond the prototype and this Bachelor thesis. The following sections will outline the most important aspects of the architecture.

5.3.1 D_ITargetable – D_Character, D_Structure, D_Item

The main components of the game environment all implement the Interface D_ITargetable. These components are D_Character, D_Structure and D_Item. All objects, which a character should be able to interact with, fall into one of these three classes. The data provided for these classes are stored in ScriptableObjects, namely D_StructureData and D_ItemData, and is

loaded to the object on game start or when a new object of the type is created. So, ideally, only three prefabs of the classes D_Character, D_Structure and D_Item would be stored centrally in a singleton D_GameMaster to be instantiated and passed the according data. This greatly reduces chaos with handling and changing multiple prefabs and improves usability, especially for non-programming developers, as they can create new objects within the Unity editor without touching any code.

The D_ITargetables are mainly used in two systems: Interactions (see 5.3.3 D_Interaction) and the grid, including pathfinding (see 5.3.4 D_CharacterControl). Otherwise it only stores and handles temporary data, as permanent data is stored in the according ScriptableObject and logic is done with Interactions.

It is important to note, that there are exceptions to this approach, mainly for time constraint reasons: First, there is no D_CharacterData as of yet. Because the D_Character with his Skills, Inventory and Maslow's is more complex, it had no clear-cut line to what is simply data and what might still change during development. In an ongoing production the interchangeable information would be stored in such a data class as well.

Next to D_CharacterData there is another exception, that is not by design, but for the lack of time resources: D_Tree and D_Fruit, introduced early in development, inherit from D_Structure and D_Item respectivly. They handle the creation and visual growth of apples. This does not fit the ideal of having only three final classes with only the data classes changing the appearance and behavior of the object. A better approach would be to have logic specific to an object run on the ScriptableObject data class, or with repeatedly called D_Interactions.

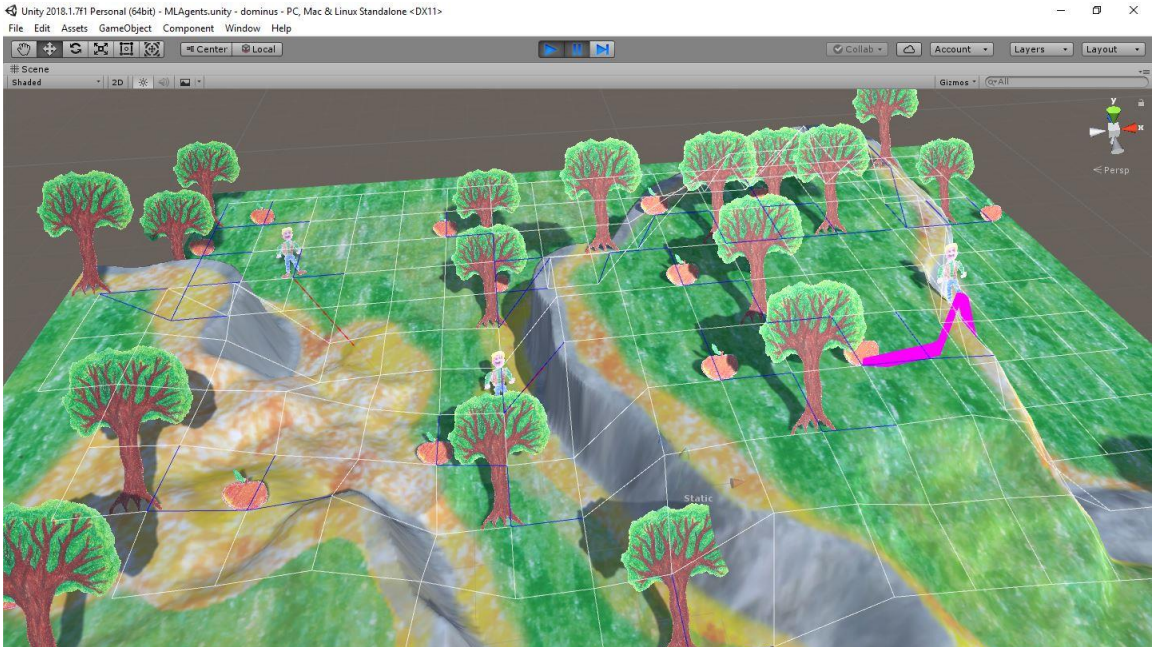

Figure 4 – Scene View of the running Prototype

5.3.4. D_Interaction

As already stated in chapter 5.1.1.7, Interactions are the most significant part in terms of game logic. Here, all the magic happens to determine the effects one object in the game world has on another. Specifically, only D_Characters can impose Interactions on other D_ITargetables. There are currently eight Interactions in Dominus that handle this:

Name	Description
Consume	The character moves to the consumable item. After the interaction time has passed, the apple gives the OnConsumptionMaslow to the character and is destroyed.
WoodCut	The character moves to the tree. After the interaction time has passed, dice equal to the characters skill in WoodCut are rolled. For each success (rolled number / 4 rounded down) the tree subtracts 1 from its integrity. When the integrity reaches 0, the tree is destroyed and a log of wood is instantiated. The interaction repeats until the tree is destroyed or the player aborts the action.
Use(Sleep)	The character moves to the house. After the interaction time has passed, the house gives the OnConsumptionMaslow to the character.
PickUp	The character moves to the item. After interaction time has passed, the item is stored in the character's inventory.
Drop	The item is removed from the characters inventory and placed on a random adjacent grid node.
BuildHouse	Three logs of wood are removed from the inventory. After interaction time has passed, the character moves to a random adjacent grid node. The complete house is instantiated on the grid node the character was originally standing on.
PlantTree	One apple is removed from the inventory. After interaction time has passed, the character moves to a random adjacent grid node. A tree is instantiated on the grid node the character was originally standing on.
OpenUI	The state of the GUI is set to what corresponds in mWindow.

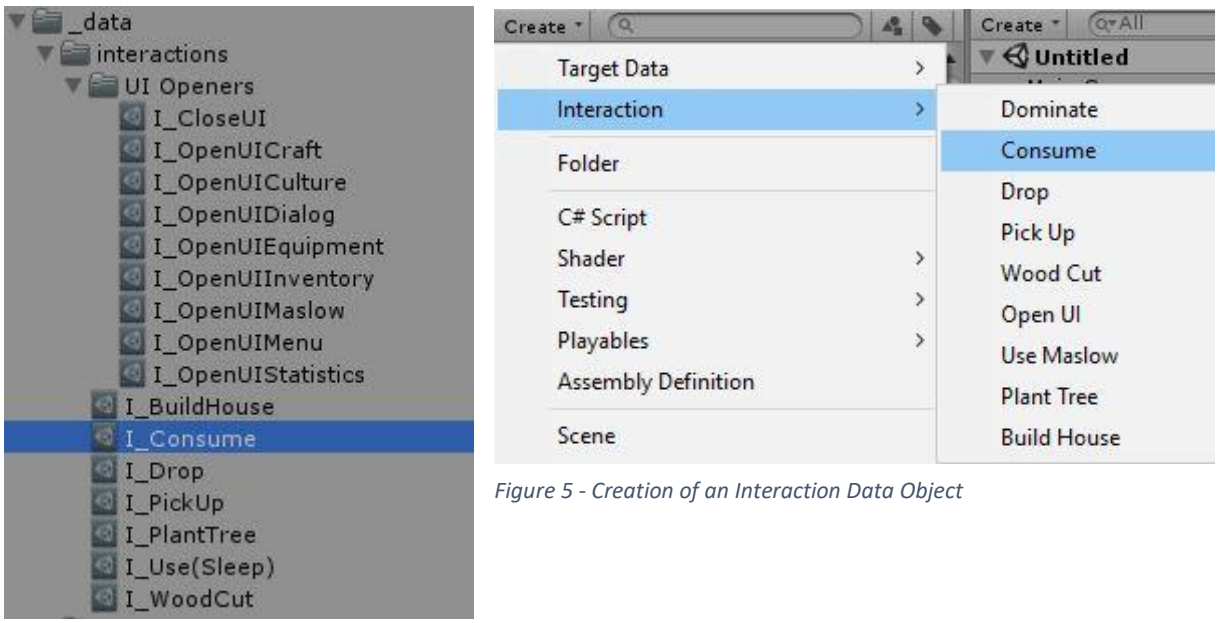

Figure 5 - Creation of an Interaction Data Object

5.3.5.1. Implementation and Command Pattern

The Command pattern, as described by Robert Nystrom, means "[...] taking some concept and turning it into a piece of data – an object – that you can stick in a variable, pass to a function, etc. [...] it's a method call wrapped in an object."[26] The concept is the Interaction. To avoid unnecessarily cluttering the unity inspector, D_Interaction derives from ScriptableObject. The D_Interaction objects reside as assets in the Unity project folder and can be easily created in the Unity editor with the Create-button.

D_Interaction serves as base-class for all logical implementations, e.g. D_InteractionConsume. D_Interaction has a virtual function "ExecuteInteraction([...])", that will be requested by a D_CharacterController, which passes the acting D_Character as "subject" and the desired D_ITargetable "target". All child-classes override this function to do with both subject and target as specified in their design. Any D_interaction, that an object should hold, is stored in a list of Possible Interactions inside the data object assigned to a D_ITargetable, as described in 5.3.3. After the logic has been implemented, only minimal setup is required to get the anticipated behavior running: The D_Interaction needs a name, that is displayed to the player, a Skill, should the logic demand a skill-check as in WoodCut, and most importantly Restriciton Flags, which determine circumstances for when and where and by whom the D_Interaction is allowed to be displayed and/or executed. These circumstances include, whether the target is in the game world or resides in an Inventory, or whether the D_Interaction may be called only by a D_Character on himself (as in OpenUI) or by the Dominator (see 5.1.1.4)

[26] Nystrom, Robert. "Game Programming Patterns" 2014. http://gameprogrammingpatterns.com/command.html Accessed Aug. 2018

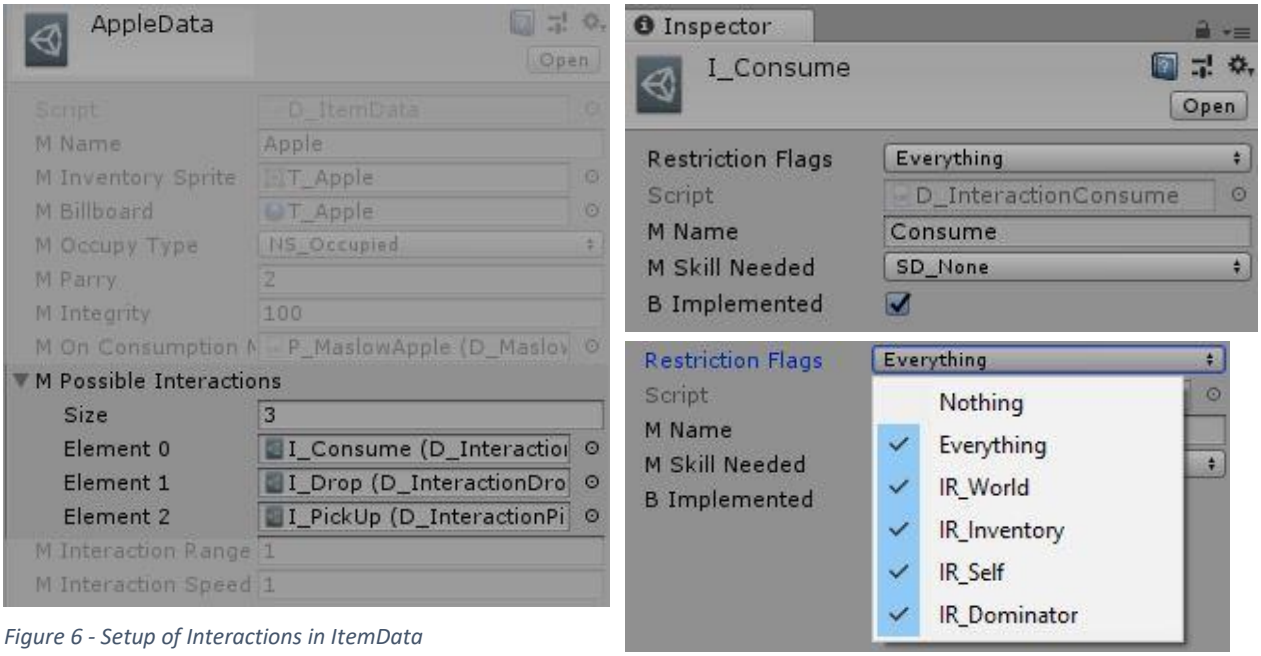

Figure 6 - Setup of Interactions in ItemData

5.3.5.2. Sequence of Interactions

When a D_CharacterControl requests execution of a D_Interaction, there is a strict sequence of steps that follow next. For example, the AI decides the best option would be to consume a close-by apple, it would pick the reference to the D_InteractionConsume and would call ExecuteInteraction() on it passing the D_Character as *subject* and the D_Item (the apple) as *target*. D_Interaction then starts its coroutine on the subject, as Coroutines cannot be started on classes, that derives from ScriptableObjects. The content of the coroutine depends on the purpose of the D_Interaction, but always follows the same structure.

The first phase of a coroutine is moving the subject to the target. The coroutine, e.g. Consume(), will first instruct the static A_Pathfinder to calculate a path from the subject to the target with the Coroutine TryPath() and will then remain dormant. When the A_Pathfinder is done calculating, it starts another coroutine MovePath() on D_CharacterControl passing a chain of waypoints and then waits for the D_CharacterControl to complete the movement. When the movement is done, A_Pathfinder finishes with its coroutine TryPath(), which signals the original coroutine Consume() to continue.

After waiting for the animation of the character for a time depending on D_Characters Interaction Speed, the coroutine carries out the code as specified in its design. In case of Consume() this means the Apple being destroyed and a Maslow being instantiated for the D_Character. When that is done, FinishInteraction() performs some cleaning and the D_Interaction is completed.

The D_Interaction can be finished early during movement and during animation, mostly due to the player interrupting it, but also when the calculated path is being blocked or when the target was unexpectedly destroyed before the interaction was completed. An internal State Machine in the D_CharacterControl keeps track of the current state of the interaction.

5.3.4.3. User Interface

The player interacts with the game world by clicking on a D_ITargetable in the world once, which prompts interaction wheel. The wheel displays all interactions, which are available at the moment. When the player picks one, the corresponding interaction sequence is issued.

Figure 7 - Graphical User Interface in Dominus

5.3.5. D_CharacterControl

D_CharacterControl is the main component to handle movement of any character. A state machine keeps track of the current status of movement. The D_CharacterControl can be ready to await new instructions, can be currently moving, can be done with the last move instruction or can be currently acting, which means waiting for a D_Interaction to finish its logic after movement is complete. Movement and acting can also be interrupted when the player decides to do another action instead, or when the interaction becomes impossible, e.g. the path gets obstructed or the targeted object becomes unavailable.

5.3.5.1 Movement, Grid and Pathfinding

The agents in the game have to find walkable paths to a desired location without bumping into other entities in the world or ignoring walls or pitfalls. Various pathfinding algorithms make this possible. In Dominus the A-Star algorithm is utilized to find a correct path from one location to the next. In order for the algorithm to work, some form of topology is required. The navigation meshes provided by Unity are problematic, as they cannot easily adapt to

changes in the environment, like a new tree being planted or a house being built, which would need a recalculation of the navigation mesh.

Instead a grid of nodes (A_Grid) was used to place entities in the game world. Nodes always have the same distance to each other in width and depth. Height of a node is calculated with samples of the heightmap of the Terrain. The number of nodes needed depends on the desired distance of nodes and the width and height of the Terrain. A_Nodes can be occupied only be a single D_ITargetable at a time. The A_Pathfinder uses these nodes to create waypoints, which hold reference to the A_Waypoint last visited and the current A_Waypoint. They also contain whether the movement is diagonal and if the node contained is the goal. When the A_Pathfinder starts the coroutine MovePath(), it passes a chain of A_Waypoints, that the D_CharacterControl will follow one A_Waypoint at a time. For convenience, also the direction of the movement is stored in a given A_Waypoint as well.

5.3.5.2 D_MLAgentsControl

This child-class of D_CharacterControl is an additional security layer before the agent can interact with the game environment. A character controlled by the machine learning algorithm can only request an interaction instead of directly issuing it. A sanity check is made prior to execution of any D_Interaction.

5.3.5.3 D_PlayerControl and D_UtilityControl

Before the MLAgents Toolkit was implemented, two child-classes of D_CharacterControl would control decision making process and behavior of any given character for the player and the Utility-based AI.

As can be seen later (in 5.4.X.) the information originally produced here, needs to be funneled through the MLAgents AI and translated in terms it can understand, in order for it to learn from the actions. So, D_PlayerControl and D_UtilityControl control have been deprecated and their logic transported to D_DecisionPlayer and D_DecisionUtility (see 5.4.2.)

5.3.6. SOS_Level

For the purpose of automated content creation, as suggested in 5.1.1.1., some means of saving created content is necessary. Additional grid space adjacent to the original grid was to be created by the Imitation Learner, which would use hand-crafted Level Designs as a Teacher. Due to time constraints the class SOS_Level for saving the level remains with limited capabilities. The functionality relies heavily on Unity's JsonUtility package and wraps any D_ITargetable in the Level as lists of strings, which are then serializable into a JSON-format.

5.4. Artificial Intelligence

Decision making processes in games can have various forms of architecture. Some of these perform well with one situation but does worse in another. Nonetheless, they all act on the same principle: Internal and external knowledge is fed into the process and the decision maker requests an action, which changes the internal and external knowledge.

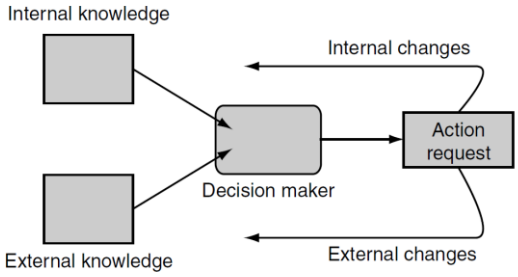

*Figure 8 – Decision Making Schematic**

Some of the most common architectures are summarized in the counter-part of this Bachelor thesis, the research project titled "Comparing Approaches to Game AI under Consideration of Gameplay Mechanics". What is not referenced there are two approaches to Game AI, which will be debated in the following Chapter: Utility-based AI and Imitation Learning.

5.4.1. Utility AI

5.4.1.1. Concept

Utility is a measure of relative satisfaction as a result of actions.[27] It differs from hard value, because it can change according to the agent and its current and future situation. An example: The value of a 20€ bill is what it is written on it, which is 20€. When such a bill is lost by a poor person, he will probably go out of his way to trace back his steps and find the money. Even though the value is the same, it means a lot to the poor person. The other way around a multi-billionaire would probably bend over to reach for the 20€ bill next to his feet, but probably would not indulge in a bigger investigation. This one bill means a lot less to the rich person than it does to the poor one. Or in other words, the poor man has high marginal utility of the 20€ bill, while the rich person has low marginal utility of the 20€ bill. It is important to look at the margin of utility, as it shows how big the impact of the according action might be.

[27] Mark, Dave. *Behavioral Mathematics for Game A.,* Charles River Media, 2009, p.111

As shown in the example, the utility of an object depends on the situation of the subject. Oftentimes these situations contain more information, which is important to make a decision,

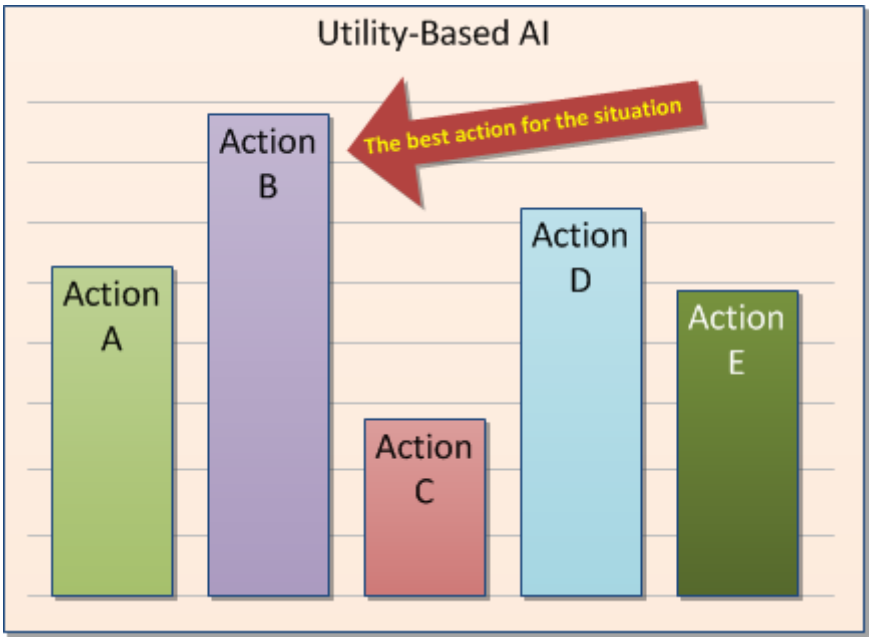

*Figure 9 – Utility-based AI Scoring Schematic**

like the distance between subject and object, the amount of satisfaction the action might give or the relation to other possible actions.

Utility-based AI scores each potential action in the game based on a combination of an agent's current needs and the ability of that action or item to satisfy that need. The agent then uses an approach common in utility-based methods and constructs a weighted sum of the considerations to determine which action is "the best" at that moment. The action with the highest score wins.[28]

The agent Sensors to perceive the game environment and to successfully determine the action scores. Each sensor checks a single value in the game world, like the distance between two entities or the current amount of health points. This data is then normalized and processed by a Utility-function in mathematical graphs of various shapes. The outputs of those functions can be combined in other utilities. This makes it possible to bias some information or groups of information, which makes them more or less important to the resulting action score. The resulting action can be a pre-defined sequence of actions.

[28] Mark, Dave. "A Culinary Guide". *Intrinsic Algorithm*, 1. Nov 2012 http://intrinsicalgorithm.com/IAonAI/2012/11/ai-architectures-a-culinary-guide-gdmag-article/ Accessed Aug. 2018

5.4.1.2. Implementation

In Dominus a Utility-based Brain creates D_AI_Actions for each potential D_Interaction, that is placed on D_ITargetable in the game environment. Then D_DecisionUtility calls Test() on each D_AI_Action, that was created in this manner, passing the subject (the issuing agent) and the target as parameters. The test includes some sanity-checks, but mainly calls ComputePoints() of the D_AI_Utility referenced in D_AI_Actions.

The connected D_AI_Utility can then either be a D_AI_UCombine, which combines and weighs two other utilities connected to them, or can be a functional D_AI_Utility, like D_AI_ULerp or D_AI_URandom, which calls RunSensor() on a given D_AI_Sensor.

The information returned from the D_AI_Sensor, a single float value, is then recursivly combined and weighted throughout the Utility tree. The value returned by the top-most D_AI_Utility is thereby the action score and is saved in the tested D_AI_Action.

The decision-making process of the Utility-based AI is finished when all D_AI_Actions are scored and the D_AI_Action with the highest score is selected.

In order to employ the Utility-based system as a teacher for the Imitation Learning process, the information saved in the selected D_AI_Action has to be translated into terms the MLAgents system can understand and is able to reproduce. The former class D_UtilityControl, that would issue the selection process of the Utility-based system, was derived from D_CharacterControl and was thereby working parallel to the MLAgents system. But the inner workings of the MLAgents Toolkit requires the decision-making process to happen in a very specific spot. That is why D_UtilityControl had to be deprecated and its logic moved to a class D_DecisionUtility, which derives from a class provided by the MLAgents Toolkit, Decision.

For time constraint reasons and for the lack of necessity in terms of testing, the prototype contains only the most basic behavior. The observable behavior of a Utility-based agent entails only to find the nearest Apple and consume it. The Utility needed for this is a simple linear interpolation, that translates the output of the Distance Sensor into utility values. The problem to find the closest Apple was sufficiently challenging to the Imitation Learning algorithm, so no further functionality was necessary.

5.4.2. Machine Learning Agents Toolkit for Unity

5.4.2.1. Concept

Unity's Technical Evangelist Alessia Nigretti describes Machine Learning as "[...]an application of artificial intelligence that provides a system with the ability of learning from data autonomously, rather than in an engineered way."[29] So, instead of a predefined algorithm determining the best action for the agent with given inputs, the Machine Learning agent is given a vector of inputs to find patterns and to predict a vector of outputs. When outcome of the prediction is wrong, the MLAgent adjusts to produce more promising prediction. This input-output mapping is repeated thousand-fold to optimize it towards the desired result. Unity's Machine Learning Agents Toolkit provides two systems, each with its distinct use case, namely Reinforcement Learning and Imitation Learning.

Reinforcement Learning, or Proximate Policy Optimization (PPO), "[...] uses a neural network to approximate the ideal function that maps an agent can take in a given state."[30]

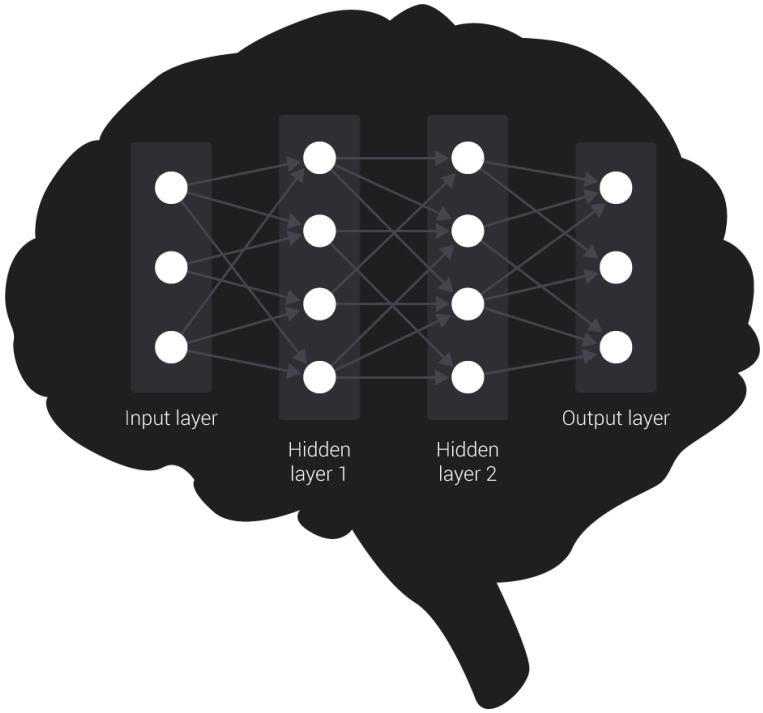

*Figure 10 – Neural Network Schematic**

A neural network resembles a nervous system. It is structured in neurons, which are divided into layers. The Input layer contains all relevant data from the game environment, while the Output layer contains the information calculated by the Hidden layers based on the Input

[29] Nigretti, Alessia. "Using Machine Learning Agents Toolkit in a real game: a beginner's guide" *Unity Blog*, 11. Dec. 2017. https://blogs.unity3d.com/2017/12/11/using-machine-learning-agents-in-a-real-game-a-beginners-guide/ Accessed Aug. 2018

[30] "Training with Proximal Policy Optimization" *Unity Technologies*. https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-PPO.md Accessed Aug. 2018

layer. Connections between neurons are weighted, which indicate the strength or weakness of the connection. Whenever the agent performs an action, the rewards and punishments, which are set by the developer, are fed back into the process. Based on the feedback the algorithm adjusts the weights between connections in a way, that reproduces outputs with desired behavior and prevents outputs with faulty behavior.

Imitation Learning, or Behavioral Cloning, "[...] uses a system based on the interaction between a Teacher agent (performing the task) and a Student agent (imitating the teacher)."[31] Different from Reinforcement Learning, Imitation Learning does not work with a reward/punishment mechanism. Instead, the Teacher agent provides samples to the Student agent, which contain input data and output data. Goal of the Student agent is to reproduce similar outputs with any given similar input. This behavior comes close to what is known as Supervised Learning, where samples, like pictures of bees and cars, are labeled with the correct answer and the Supervised Learner has to predict the label of any given sample. In context of a game, the picture resembles the game state, while the correct label is an action by the Teacher, e.g. a button press. When the prediction is wrong, for example when in a given situation the Student agent would turn left but the Teacher agent turns right, the weights of the connections in the neural network are adjusted again to account for the error.

The science and mathematical calculations behind neural networks are vast and complex. An extensive dive into the matter would require prolonged studies and is therefore beyond the scope of this Bachelor thesis. But Unity's technicians realized the inaccessibility of the matter as well, which is why they created the MLAgents Toolkit in the first place. That way, developers of wide ranges of skill levels can employ such a system without needing a degree in scientific studies. Compared to what a deep understanding of the algorithm entails, the implementation of the MLAgents Toolkit is relatively simple.

Unity's Machine Learning Agents Toolkit is built on-top of the open-source library TensorFlow, which contains the algorithms and is responsible for the training of agents, but does not provide a native C# API. Therefore, it runs outside the Unity Editor and is connected to it via an External Communicator. For the purpose of this Bachelor thesis the inner workings of TensorFlow are treated as a black box. The main components of the toolkit are the Academy, Brains and Agents.[32]

---

[31] Nigretti, Alessia. "Imitation Learning in Unity: The Workflow" *Unity Blog,* 24. May 2018. https://blogs.unity3d.com/2018/05/24/imitation-learning-in-unity-the-workflow/ Accessed Aug. 2018

[32] Juliani, Arthur. "Introducing: Unity Machine Learning Agents Toolkit" *Unity Blog*, 19. Sep. 2017. https://blogs.unity3d.com/2017/09/19/introducing-unity-machine-learning-agents/ Accessed Aug. 2018

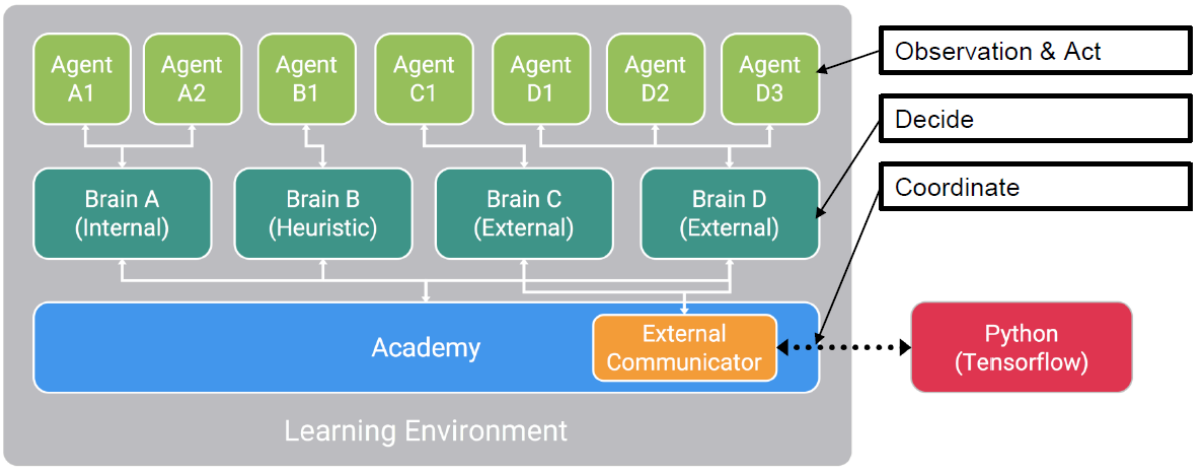

*Figure 11 – Unity Machine Learning Agents Schematic**

Agents correspond to the characters in the game. They are responsible for collecting observations and carrying out actions. Each Agent has a Brain, which make the decisions and issues a corresponding action based on the observations of the Agent. Several Agents can have the same Brain. Brains are set to one of four modes:

- External – Action decisions are made using TensorFlow
- Internal – Actions decisions are made using a trained model embedded into the project
- Player – Action decisions are made using player input.
- Heuristic – Action decisions are made using hand-coded behavior.

The Academy conducts communication between External Brains and TensorFlow. Also, it is responsible for coordinating the environment for a training episode.

5.4.2.2. Implementation

The D_Academy has three important functions, that override the virtual functions provided by the MLAgents Toolkit: InitializeAcademy() is responsible for setting up the environment and is called once when the game starts. Here a terrain is picked from a list and instantiated, a A_Grid is set up for it and agents are being created for each brain, that should be trained. Additional characters, that do not require a machine learning brain, can be created as well.

AcademyReset() is called when every active Agent reached the goal of the training scenario or when the step count (with each FixedUpdate() call being a step) for a training episode exceeds the maximum count per episode, which is set in the base class of the Academy. Here, the content of the current training episode is shaped by placing Trees and additional structures and items at random locations on the Grid. Prior to that, all remnants of the old training episode have to be destroyed.

AcademyStep() is called each step, so on each FixedUpdate() call, and can be used to orchestrate the environment, possibly placing additional Trees when there are not enough around. In Dominus nothing of the sort was necessary, though.

The challenging part to implement MLAgents becomes apparent in D_Agent. AgentReset() prepares the agent for a new training episode when the Academy resets the environment. D_Agent has two major responsibilities: CollectObservations() translates all necessary information in the game world into terms the Machine Learning algorithm can use, which corresponds to the Input layer of the Brain. This means that all information has to be available as floating-point numbers ideally ranging between -1.0 and 1.0. The other way around, AgentAction() translates the computations of the Brain, so the Output layer, to actions in the game. The raw output is only available as floating-point numbers ranging from -1.0 to 1.0 as well. AgentAction() also gives opportunity to reward or punish the Brain for its decision when using Reinforcement Learning. Careful considerations had to be made for setting up the collection of observations and the translation of the Brains output into game actions.

Next to its own position on the grid, its current ready-state and its Happiness, the Agent considers a maximum of 50 Targetables when collecting observations. The number of observed Targetables can vary and is padded with zeros for any missing entity, as the observation vector must always contain the same number of elements.[33] The properties important to the Agent are the Targetables' positions and the Interactions available on them. The Agent only considers Interactions, that are listed to him as known. It is important to note that the Interactions listed as known must be ordered in a specific way to make D_DecisionUtility able to communicate with the Agent. The available Interactions are encoded in a one-hot style, so each of them has a corresponding entry in the observation vector with a value of either 1.0, when the Interaction is available, or 0.0, when it is unavailable.
When the Agent knows about all Interactions included in the prototype, excluding OpenUI and including plain movement, the input vector has a size of 454 observations to keep track of. A simple version, in which only movement and Consume are available, reduces the amount to 154 observations. This number is required to be correctly set in the Brain-class.

[33] "Continuous Vector Observation Space: Feature Vectors" *Unity Technologies*. https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Design-Agents.md#continuous-vector-observation-space-feature-vectors Accessed Aug. 2018

The vector of floating-point numbers received by AgentAction() has a fixed size as well. A full support of all Interactions has an action vector of size 10. The action vector in the simple approach contains only 4 floating-point numbers. The first and second number always corresponds to normalized coordinates of a node on the grid. Should any other entry surpass a minimum threshold, the corresponding Interaction is tested for being viable for a possibly given Targetable on the node. When the Interaction is viable, it executed as stated in 5.3.5.2. Otherwise another decision is requested from the Brain.

D_DecisionUtility ignores all observations done in the Agent, and instead collects its information as described in 5.4.1.2. The resulting action cannot be directly executed, though. Instead it first has to be translated and funneled through the MLAgents system to make it possible for the Utility-based AI to function as a Teacher for the Brain. Therefore, an action vector of floating-point numbers is created containing the necessary information, that is instantly translated back into an Interaction within AgentAction(). D_DecisionUtility is placed on the same GameObject as the Brain and requires the Brain to be set to “Heuristics” mode.
D_DecisionPlayer is not implemented due to time constraints and complications in the implementation. As D_DecisionUtility suffices to show the Imitation Learning capabilities, this functionality is only a nice-to-have. For purposes of future presentations, this class will be implemented in the aftermath of this thesis.

6. Evaluation

6.1. Choices in technology

6.1.1. Choices in the prototype architecture

There are four complicated, high-maintenance systems in the prototype:

- Interactions (Command Pattern)
- Utility-based AI
- Pathfinding
- Machine Learning Agents

A simpler prototype would have sufficed to test the Imitation Learning system, as it would have been less prone to bugs. On the other hand, there would have been less insight won, were the prototype as simple as the examples, that are shipped with the download of the Machine Learning Agents plugin, e.g. the Banana Collector. These insights include problems with the amount of observations and actions, that an agent can handle and the limitations in handling indirect controls of the agent.

6.1.2. Choice in Artificial Intelligence

While the thesis was originally planned to use only Utility-based AI, it became clear that to make it work in a learning fashion a whole new system, research and development would have been necessary to create similar results to the Imitation Learning algorithm, which simply was not feasible. The technology of Imitation Learning was only presented and released late in the development of the prototype. A lot of code had to be refactored to accommodate to the requirements of the new technology. Also picking up the rather complicated technology while working on the prototype stretched the time resources.

It was the right choice to switch from Utility-based AI to the Machine Learning system, but the prototype does not show the work invested in the former. The Utility-based AI system was not explicitly needed for Imitation Learning to work, but it was very convenient for training sessions that lasted for hours.

The switch between both systems also brought up issues, that would not have been as obvious in case of the prototype being cut only to the requirements of Machine Learning.

6.2. Direct control versus indirect control

There is a difference between direct and indirect control of characters in the game world. Direct control means that the player pushes the “Walk Left”-button and the character walks to the left, relative to its own position. With indirect control, on the other hand, the character is given a position, that it should walk towards, that can be close to him or far away. It then calculates a path, which can initially lead the character in the opposite direction to avoid collision with objects in the path. Good examples for the difference between the two are the PC version of Diablo3 [Blizzard. 2012], where the of the game utilizes indirect control with the player clicking where the character should go, and the console version, where the game needs to accommodate to the input with joysticks with the player controlling the character directly. Utility-based AI does not explicitly require indirect control to properly function, but it is common to use it in such a way. When the objects in the game environment produce the actions available to the character, the decision making system only has to decide between the best actions at the time. The movement and pathfinding system can be easily separated from the decision making process.

Any examples shipped with the Machine Learning Agents Toolkit rely on direct control. It is not explicitly stated that indirect control is not possible. The Player-mode of the Brain suggests that only direct control is intended, as there can be buttons mapped to conveniently connect the Machine Learning Agents system with Unity’s Input system.

The workaround found in Dominus is to trick the system in believing the player input to be some kind of a version of a Brain set to Heuristic-mode. That is why there is D_DecisionPlayer, which is supposed to work similarly to D_DecisionUtility. Due to time constraints, this solution is only theoretical and still requires a proof of concept.

6.4. Keeping track of all Targetables

With Utility-based AI keeping track of all Interactions is not a problem, in theory. The number of Interactions listed for scoring can range from only a few to hundreds and thousands. When there is a new Interaction to keep track of, it is simple to add it to the AI system compared to what is necessary with Machine Learning. The only limitation for Utility-based AI is efficiency when the scoring of Interactions relies on a lot of Utility-functions. This can be optimized by limiting the Targetables to keep track of to the closest ones.

With Machine Learning there are problems that can not be fixed as easily. The Input and Output layers are required to be set to a fixed size. This is why padding the Input layer with zeros was necessary to have potential room for additional Targetables to keep track of. The suggestion by the developers of the Machine Learning Agents Toolkit of Interactions being kept in a one-hot style poses another problem: Next to the maximum amount of trackable Targetables being fixed, also the number of trackable Interactions cannot easily be increased. When there is a new Interaction for the Machine Learning System to recognize, the Input vector has to be resized. This requires a new Brain to be trained from the ground up. Obviously, this is not feasible for a game design as suggested with Dominus, where the game's content, including Interactions, should grow on a regular basis.

Another workaround would have to be found to join the benefits of Imitation Learning with the necessity of growing content.

6.5. Testing Proximate Policy Optimization

Reinforcement Learning is not a major concern for the prototype, because the Agents were never meant to play the game in an optimal way, but in a human way. Nonetheless, testing the game environment with only one Agent, that uses Proximate Policy Optimization (PPO) for training, gave insight over possible misconceptions and bugs in the code.

It is interesting to note that punishing the Agent when it tried to issue an impossible Interaction resulted in the Agent not trying to get to the closest Apples, but instead made sure to walk long distances to not be forced to make another decision, which was probable to punish it again.

6.6. Testing Imitation Learning

The example of Antigraviator shown in Unity's presentation for Imitation Learning claims the technology to be able to imitate player behavior to being undistuingishable within minutes. The implementation of Imitation Learning in Dominus comes nowhere close to that. After six hours of training the observable behavior of the Student Agent still shows it standing still for large amounts of time, even though the task to solve was simplified significantly to only find and consume the nearest Apple. There are many possible reasons for this:

One possibility is for the Imitation Learner not being able to handle huge amounts of information in the Input layer. The Antigraviator-example needs 20 observations to function. In comparison, Agents in Dominus need to keep track of 454 observations. This highly ambitious number could be reduced to 94 observations by limiting the trackable Targetables to the 10 closest ones. This would render the technology unusable for a game like Dominus, where many and far away objects in the game have importance to the character's behavior. With the length of the training session in mind, this explanation seems implausible.

Another more plausible reasoning is that the Imitation Learner is not only recording the behavior of the Teacher when it actually makes a decision, but also when the Teacher is moving or acting with an object. The implementation in Dominus made sure to make use of the RequestDecision()[34] functionality provided by the toolkit. This essentially means that computations for the Teacher Brain are only made when necessary, so after another interaction is completed when the Agent needs something else to do. This explanation can be retraced to the observable behavior of the Student Agent, where the action vector is changing in intervals ranging from a few seconds to several minutes. A possible solution would be to start recording the Teacher's behavior prior to any decision making and to pause recording instantly after an Interaction is issued and the Agent starts moving. Due to time constraints this possible solution was not investigated.

[34] "On Demand Decision Making" *Unity Technologies.* https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Design-Agents.md#on-demand-decision-making Accessed Aug. 2018

6.7. Necessity of External communicator

The need for an external communicator and the preparations required to set up the Machine Learning environment, including installing Python, makes the Machine Learning Agents Toolkit unsuitable for use in a commercial version of Dominus. This could only be fixed, if it was possible to ship the TensorFlow-part with the game's executable. Whether the developers of the toolkit plan on making this possible, is currently unknown. With the completion of this Bachelor thesis, there are strong intentions to make a request towards Unity's development team to have a look into possibilities of integrating the Machine Learning Agents Toolkit fully into the Unity Editor.

7. Future Work and Outlook

The investigations done in this Bachelor thesis are far from covering all aspects of Imitation Learning. The pursuit of a solution to the initial problem gave insight to what is important when considering Machine Learning for a game. The problems appearing throughout the development are intriguing. Further research, creative workarounds and alternative solutions are well within reach for additional work.

One aspect of possible future work is deepening the understanding of hyperparameters to boost the efficiency of the learning process.

A bigger impact on the implementation of Machine Learning is expected with further investigations and development in the setup of the environment, the observation vector and the action vector.

The most interesting idea, though, is to completely rework the decision making system to meld the Utility-based AI and the Machine Learning algorithm into a single system with distributed responsibilities for each of them. Instead of using Utility-based AI as a Teacher to create basic behavior and Machine Learning fully reproducing the behavior making the Utility-based AI obsolete in the long run, a cooperation would emphasize the strength of both systems. In this idea, the Utility-based AI is responsible for collecting all Interactions while the Machine Learning algorithm becomes important only for scoring the collected Interactions, shifting the weights in the Utility-mechanism to accommodate for the player's choices, when the Utility-based AI would have chosen a different action.

As can be seen, there are many possibilities for future research of Machine Learning within the context of a game development.

8. List of References

Amazon Web Services	2018. https://aws.amazon.com/ Accessed Aug. 2018
Baron, Sean	"Cognitive Flow: The Psychology of Great Game Design" Gamasutra, 22. Mar. 2012 https://www.gamasutra.com/view/feature/166972/cognitive_flow_the_psychology_of_.php Accessed Aug. 2018
Bartle, Richard	"Hearts, Clubs, Diamonds, Spades: Players who suit MUD" Apr. 1996. http://mud.co.uk/richard/hcds.htm Accessed Aug. 2018
Berges, Vincent-Pierre	"Democratize Machine Learning" Unite Berlin - Day 3 Sessions Livestream 21. Jun. 2018 https://www.youtube.com/watch?v=a768FLX9bRc&t=2h30m3s Accessed Aug. 2018
Deeplearning4j	2017. https://deeplearning4j.org/ Accessed Aug. 2018
Faggella, Daniel	"What is Machine Learning?" TechEmergence, 2. Sep. 2017 https://www.techemergence.com/what-is-machine-learning/ Accessed Aug. 2018
Green, Ollie	"Minecraft: Hunger Games is the best Battle Royale Game" GameByte. 10. May 2018 http://www.debate.org/opinions/minecraft-hunger-games-or-fortnite-battle-royale Accessed Aug. 2018
Juliani, Arthur	Democratizing Deep Learning with Unity ML-Agents Unity at GDC 28. Mar. 2018 https://www.youtube.com/watch?v=YsEDv13W1RI&t=12m20s Accessed Aug. 2018
Juliani, Arthur	"Introducing: Unity Machine Learning Agents Toolkit" Unity Blog, 19. Sep. 2017. https://blogs.unity3d.com/2017/09/19/introducing-unity-machine-learning-agents/ Accessed Aug. 2018
Mark, Dave.	Behavioral Mathematics for Game AI. Charles River Media, 2009
Mark, Dave.	"A Culinary Guide". Intrinsic Algorithm, 1. Nov 2012 http://intrinsicalgorithm.com/IAonAI/2012/11/ai-architectures-a-culinary-guide-gdmag-article/ Accessed Aug. 2018
McLeod, Saul	"Maslow's Hierarchy of Needs" SimplyPsychology, 2018. https://www.simplypsychology.org/maslow.html Accessed Aug. 2018
Microsoft Azure	2018. https://azure.microsoft.com/ Accessed Aug. 2018
Nigretti, Alessia	"Imitation Learning in Unity: The Workflow" Unity Blog, 24. May 2018. https://blogs.unity3d.com/2018/05/24/imitation-learning-in-unity-the-workflow/ Accessed Aug. 2018

Nigretti, Alessia	“Using Machine Learning Agents Toolkit in a real game: a beginner’s guide” Unity Blog, 11. Dec. 2017. https://blogs.unity3d.com/2017/12/11/using-machine-learning-agents-in-a-real-game-a-beginners-guide/ Accessed Aug. 2018
Nintendo	“Minecraft: Nintendo Switch Edition” 12. May 2017. https://www.nintendo.de/Spiele/Nintendo-Switch/Minecraft-Nintendo-Switch-Edition-1214741.html Accessed Aug. 2018
Nystrom, Robert	“Game Programming Patterns” 2014. http://gameprogrammingpatterns.com/command.html Accessed Aug. 2018
OpenNN	2018. http://www.opennn.net/ Accessed Aug. 2018
Oracle	“Oracle Data Mining” http://www.oracle.com/technetwork/database/options/advanced-analytics/odm/overview/index.html Accessed Aug. 2018 Accessed Aug. 2018
Prometheus Games	“Savage Worlds - Gentleman’s Edition Revised - Probefahrt” 2013. https://www.prometheusgames.de/download/savage-worlds/SW-GER-Probefahrt-v1.11-web.pdf Accessed Aug. 2018
Staats, David.	”Designing Cooperative Gameplay Experiences“ Gamasutra, 15. Dec. 2015. https://www.gamasutra.com/blogs/DavidStaats/20151221/261927/Designing_Cooperative_Gameplay_Experiences.php Accessed Aug. 2018
TechEmergence	“About TechEmergence” https://www.techemergence.com/about/ Accessed Aug. 2018
TensorFlow	https://www.tensorflow.org/ Accessed Aug. 2018
TensorFlow	“TensorBoard: Graph Visualization” 19. Jul 2018. https://www.tensorflow.org/guide/graph_viz Accessed Aug. 2018
Unity Technologies	Unity Machine Learning Agents Toolkit. 2018. https://github.com/Unity-Technologies/ml-agents Accessed Aug. 2018
Unity Technologies	“Continuous Vector Observation Space: Feature Vectors” https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Design-Agents.md#continuous-vector-observation-space-feature-vectors Accessed Aug. 2018
Unity Technologies	“Issue #962: Citation of the project.” 10. Jul. 2018. https://github.com/Unity-Technologies/ml-agents/issues/962 Accessed Aug. 2018
Unity Technologies	“ML-Agents Challenge 1” 31. Jan. 2018 https://connect.unity.com/challenges/ml-agents-1 Accessed Aug. 2018
Unity Technologies	“On Demand Decision Making” https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Design-Agents.md#on-demand-decision-making Accessed Aug. 2018

Unity Technologies	"Training with Proximal Policy Optimization" https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-PPO.md Accessed Aug. 2018
Wwcftech	"Minecraft Sales Reach 144 Million Across all Platforms; 74 Million Monthly Players", 1. Jan. 2018 https://wccftech.com/minecraft-sales-144-million/ Accessed Aug. 2018

9. List of Figures

Figures marked with an asterik (*) are taken from the following sources:

Figure 1 - Nigretti, Alessia. "Imitation Learning in Unity: The Workflow" *Unity Blog,* 24. May 2018.
Figure 2 - McLeod, Saul. "Maslow's Hierarchy of Needs" *SimplyPsychology*, 2018.
Figure 7 – Millington, Ian. *Artificial Intelligence for Games*. 2nd ed., Morgan Kaufmann Publishers, 2009.
Figure 8 – Mark, Dave. "AI Architectures: A Culinary Guide". *Intrinsic Algorithm,* 1. Nov. 2012.
Figure 9 – Nigretti, Alessia. "Using Machine Learning Agents Toolkit in a real game: a beginner's guide" *Unity Blog*, 11. Dec. 2017.
Figure 10 – "ML-Agents Toolkit Overview" *Unity Technologies.* https://github.com/Unity-Technologies/ml-agents/blob/master/docs/ML-Agents-Overview.md Accessed Aug. 2012

Erklärung:

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel und Quellen verwendet habe.
Soweit ich auf fremde Materialien, Texte oder Gedankengänge zurückgegriffen habe, enthalten Ausführungen vollständige und eindeutige Verweise auf Urheber und Quellen.
Alle weiteren Inhalte der vorgelegten Arbeit stammen von mir im urheberrechtlichen Sinn, soweit keine Verweise oder Zitate erfolgen.
Mir ist bekannt, dass ein Täuschungsversuch vorliegt, wenn die vorstehende Erklärung sich als unrichtig erweist.

_____ _____

Ort, Datum Unterschrift

Erklärung zur Archivierung:

Bitte zutreffendes ankreuzen:

- Mit der Archivierung der gedruckten Abschlussarbeit in der Bibliothek

  bin ich einverstanden.

- Mit der Archivierung der gedruckten Abschlussarbeit in der Bibliothek

  bin ich nicht einverstanden.

  Begrünung:

  - Die Arbeit ist gesperrt, da sie in einem Betrieb durchgeführt wurde und ihr Inhalt ausdrücklich durch diesen gesperrt ist. (Vgl. ABPO § 18 (9))
  - Persönliche Gründe

_____ _____

Ort, Datum Unterschrift