TÉCNICO LISBOA

Social Agents in *Minecraft*

Carlos Alberto Azevedo Marques

Thesis to obtain the Master of Science Degree in

Engenharia Informática e de Computadores

Supervisor: Prof. Rui Filipe Fernandes Prada

Examination Committee

Chairperson: Prof. António Manuel Ferreira Rito da Silva

Supervisor: Prof. Rui Filipe Fernandes Prada

Member of the Committee: Prof. Joana Carvalho Filipe de Campos

November 2022

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

Throughout my work on this thesis, I have had a lot of setbacks. I experienced burnout as I have never felt before, I lost my mother and I had a fair share of mental health issues. Despite all of that, this document is finished.

First off, I will thank my grandmother Lurdes, who pays my tuition and allowed me to continue my studies and pursue my dreams. Without her emotional and financial support, I would not have finished this.

I want to thank my supervisors, Professor Rui Prada and Diogo Rato, not only for guiding me through this project and sharing their expertise but also for having the patience of saints, as even when my progress was slow or came to a halt, they never stopped supporting me or got upset.

I also want to thank my friend Gabriel Freitas, who also struggled through his dissertation this year, and always encouraged me through my struggles.

I would be remiss if I did not mention the entirety of the Gamedev Técnico and Laboratório de Jogos communities, who have been friends since I joined them. The many projects we worked on together offered me a respite from my troubles and made me enjoy my final year as a university student even more.

And speaking of communities, I thank the members of both the PrismarineJS and Minecraft on Docker Discord servers, for helping me out when I hit roadblocks.

And last, but certainly not least, I want to thank close friends like Lisa, Sam, Mara, Ruhan, João, Tomás, and many more, who supported me and helped me keep myself sane and happy throughout the year.

Resumo

Os jogos de video continuam a expandir em escala, oferecendo aos jogadores mundos cada vez maiores para explorarem e ficarem imersos. E com mundos maiores, vem a necessidade dos designers popularem-nos com personagens interessantes, que pareçam parte de mundo e com quem o jogador pode criar laços. Contudo, o esforço necessário para individualmente criar cada personagem para atingir esta profundidade desejada não é fazível.

O objetivo da nossa pesquisa é para remover a necessidade de criação individual e oferecer uma framework onde os designers ditam com personagens se devem comportar, mas não se têm de preocupar com as minúcias da tarefa. Pretendemos aplicar um modelo que permite o lançamento de uma rede de larga escala de personagens, que se comportam como membros de uma sociedade, e têm relações interpessoais entre elas.

Para testar a nossa framework, criámos duas vilas de agentes no Minecraft, uma muito expressiva e sociável, e outra não tanto. Fizemos os nossos sujeitos jogar Minecraft, e seguir a lenhadora da vila social, que trabalhava com o lenhador da outra vila, durante um dia inteiro. Isto permitiu aos sujeitos observar o comportamento de cada agente e contrastá-los.

Palavras-chave: Non-Player Character (NPC), Jogos de video, Sociedade, Esforços de Autoria

Abstract

Games keep expanding in scope, providing the player with increasingly bigger worlds to explore and fully immerse themselves in. And with bigger worlds, comes the need for designers to populate them with interesting characters, that feel like part of the world and that the player can forge a bond with. However, the effort required to individually author each character to reach this desired depth is unfeasible.

Our research goal is to remove the need for individual authoring and provide a framework where designers dictate how characters should behave, but still do not need to concern themselves with the minutia of the task. We aim to apply a model that allows for a large-scale character network to be deployed, with characters that behave like they are members of society, and have interpersonal relationships with each other.

In order to test our framework, we created two villages of agents in Minecraft, one very expressive and sociable, and one not as much. We had our subjects play Minecraft, and follow the lumberjack of the sociable village, who worked with their other village counterpart, for a full day. This allowed the subjects to observe each agent's behavior and contrast them.

Results were mixed, but promising, with agents doing great in many of the parameters set for believability, but having a lot of technical problems.

Keywords: Non-Player Character (NPC), Authoring Effort, Society, Video Games

Contents

Acknowledgments . . . iii
Resumo . . . iv
Abstract . . . v
List of Figures . . . ix
List of Listings . . . x
Acronyms . . . xi

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem . . . 2
1.3 Objective . . . 2
1.4 Thesis Outline . . . 2

2 Background **4**
2.1 *Minecraft* . . . 4
2.1.1 Uses in Research . . . 4
2.2 *Prismarine* . . . 5
2.2.1 Minecraft data . . . 6
2.2.2 *Mineflayer* . . . 6
2.3 *Docker* . . . 7

3 Related Work **8**
3.1 Mods . . . 8
3.1.1 Socialcraft . . . 8
3.1.2 Minecraft Comes Alive . . . 9
3.2 Social Systems Architectures . . . 10
3.2.1 *Façade*: An Experiment in Building a Fully-Realized Interactive Drama . . . 10
3.2.2 *Versu* - A Simulationist Storytelling System . . . 12
3.2.3 *Prom Week*: Designing past the game/story dilemma . . . 13
3.2.4 A Simple and Method for Evolving and Large Character and Social Networks . . . 14

4 Implementation **17**
4.1 Core Concepts . . . 17
4.1.1 Agent . . . 17
4.1.2 Database . . . 18
4.1.3 Location . . . 19
4.1.4 Practice . . . 20
4.1.5 Social Practices . . . 23
4.1.6 Jobs . . . 27
4.1.7 Context . . . 28
4.1.8 Identities . . . 28
4.2 Agents' Main Loop . . . 29
4.3 Deployment . . . 29
4.3.1 Bots . . . 29
4.3.2 Server . . . 31
4.3.3 File Structure . . . 33

5 Evaluation **34**
5.1 Evaluation Goals . . . 34
5.1.1 Believability of the agents . . . 34
5.1.2 Sense of society . . . 35
5.1.3 Flexibility of the framework . . . 35
5.2 Scenario . . . 35
5.2.1 Test Environment . . . 35
5.3 Problems . . . 38
5.3.1 Scope Problems . . . 38
5.3.2 Technical Problems . . . 39
5.4 Test Procedure . . . 39

6 Results **40**
6.1 *Minecraft Experience* . . . 40
6.1.1 Overall Experience . . . 40
6.1.2 Multiplayer Experience . . . 41
6.2 Believability . . . 41
6.2.1 Awareness . . . 41
6.2.2 Behaviours Understandability . . . 42
6.2.3 Personality . . . 43
6.2.4 Visual Impact . . . 43
6.2.5 Predictability . . . 44
6.2.6 Behaviour Coherence . . . 45
6.2.7 Sociability . . . 46

6.2.8 Final Tally 46
6.3 Sense of Society 47
6.3.1 Daily Routines 47
6.3.2 Society 48
6.4 Social Differences 48
6.5 Assorted Feedback 49

7 Conclusions **50**
7.1 Achievements 50
7.2 Limitations 51
7.3 Future Work 51

Bibliography **53**

A Questionnaire **55**

List of Figures

3.1 *Talk of the Town* world simulation algorithm 15

4.1 A diagram exemplifying how the Greet practice works. Agent A decides to greet Agent B. When Agent B hears it, he runs his *accepts* function and agrees to it, marking that he is socializing and his social partner is Agent A. He replies, and Agent A then undergoes the same process. 24

5.1 Village A 36
5.2 Village B 37
5.3 Casey working with Alex. Note that Casey has greeted Alex. 37

6.1 The distribution for the level of experience with Minecraft 40
6.2 The distribution for the level of experience with Minecraft multiplayer 41
6.3 The distribution of agent awareness 42
6.4 The distribution of agent understandability 42
6.5 The distribution of agent personality 43
6.6 The distribution of agent visual impact 44
6.7 The distribution of agent predictability 44
6.8 The distribution of agent coherence 45
6.9 The distribution of agent sociability 46
6.10 The distribution of agent adherence to daily routines 47
6.11 The distribution of agent belongingness in society 48

List of Listings

3.1 Lumberjack identity code . . . 8
4.1 An example of an Agent as defined if JSON . . . 17
4.2 An example of a house as defined in JSON . . . 19
4.3 ChopWood.js . . . 21
4.4 Greet.js constructor . . . 25
4.5 Greet.js getSalience . . . 26
4.6 Greet.js accepts . . . 26
4.7 Definition of Lumberjack in jobDefinitions.js . . . 27
4.8 Attributes of context.js. The database is included as it is the only way to pass it on to the practices for a consultation currently. . . . 28
4.9 Salience Rules in the constructor of friend.js . . . 28
4.10 Socialcraft Dockerfile . . . 31
4.11 docker-compose.yml . . . 31

Acronyms

AABB Axis Aligned Bounding Box

ABL A Behaviour Language

AI Artificial Intelligence

API Application Program Interface

CiF Comme il Faut

CMD Command

DM Dungeon Master

EULA End User License Agreement

GDMC Generative Design in Minecraft Competition

ID Identification

JSON Java Script Object Notation

KB Knowledge Base

NPC Non Player Character

npm Node Package Manager

PVP Player Versus Player

RPG Role Playing Game

UI User Interface

YAML YAML Ain't Markup Language

Chapter 1

Introduction

1.1 Motivation

Game worlds keep getting bigger and bigger, as hardware limits keep getting pushed. Exploration has always been a big component of gaming, going back to the 1980s with Atari's *Adventure* [1] and Nintendo's *The Legend of Zelda* [2], which featured large worlds for the player to explore, instead of single screen arenas, or linear scrolling levels.

Of course, exploration was already a part of gaming before they became electronic, as *Dungeons & Dragons*, and other tabletop role-playing games (RPGs) already let its dungeon masters (DM), players who run the game, create their own maps for the party, the players who created characters to roleplay as in order to play the game. These maps were populated by characters created and performed by the DM, non-player characters (NPCs), and they could give out quests, hints, rewards, or just provide conversation to enhance the world [3].

Video games owe a lot to the games that came before them, and as technology got better, games could have NPCs, just like tabletop games, but limited of course, as a computer script will always be limited in creativity and adaptability against a person. Filling these worlds with interesting NPCs was already a challenge, but the worlds keep getting bigger, and the demands of the players increase. NPCs started out as characters that stood in one spot, repeated the same dialog when approached, and had very limited possibilities of interaction. Nowadays, games can have whole cities populated with NPCs with various voice lines, simulating a bustling sidewalk; games can have NPCs that react in subtle ways to what the player does; games can have thousands of NPCs interacting together, as part of a social group.

While what we can do with NPCs expands, so does the workload required for creating them. Scripting all their interactions and coding them into the game is a hefty process. So much so, game companies keep looking for ways to speed up the process, by leaving it up to procedural generation. As such, there have been attempts to create frameworks that would allow developers to achieve it. Our project is just that.

1.2 Problem

Game designers have less control over the content in the game the more it is automatically generated. If the designers create something from scratch, they can control its every aspect and will have the knowledge of how to adjust it to address feedback. If something is generated automatically, the designer does not have much control over what is generated, and making alterations can be difficult, especially without the full knowledge of how something was generated.

When it comes to creating a society of NPCs, if they were wholly authored by designers, it would take a lot of work to code every behavior and situation to the designers' liking. But if it was wholly automated, which should actually be impossible, then the designers would have no control over the end results.

A hybrid solution is required, where it is possible to deploy a lot of NPCs without much work, but there is still plenty of customization allowed by the designers, which is the end goal of this thesis.

1.3 Objective

The objective of this work is to create more robust NPCs. By this we mean NPCs that exhibit a larger scope of social affordances, that can develop relationships with each other in a meaningful way, and that have interactions with the player and the world that do not feel overtly stock or scripted. To achieve this we will expand upon the preexisting SocialCraft framework and use it to endow agents with social behaviors like daily social routines, social roles, and interpersonal relationships. The NPCs will have an opinion of each other. The player must see the agents as a thriving society.

Our main way of achieving this is by taking the limited concept of identity, as explained in section 3.1 and expanding it into several other concepts: practices, jobs, and identities (though these will be different), each with a more specific role.

As such, this thesis has three main goals:

1. Deploy agents into Minecraft

2. Create robust NPCs

3. Provide a versatile framework to accomplish the other goals

1.4 Thesis Outline

In this document, we will first take a look at what is *Minecraft*, and why it was chosen. We will also take a look at *Prismarine*, which served to deploy the bots and code their behavior, and *Docker* which hosted the servers and the bots.

After, we will take a look at the work that has inspired ours, not only previous *Minecraft* modifications, but also research papers on NPC generation.

Then we will explain in detail how the agents' decision-making works, the key concepts of our framework, and how each of them works and interacts with each other. Also, we will talk about how the server was deployed using *Docker* and the overall file structure of the project.

Following this, we will discuss our evaluation goals, as in, what we wanted to discover from our tests, how the user test was designed and performed, as well as the setbacks we faced during them. This is followed by an analysis of the results.

Finally, we weigh the accomplishments versus the failings of our project and suggest future work to be done on it.

Chapter 2

Background

Our framework could not be tested unless there was a scenario that could be created for users to interact with. Hence, Socialcraft was developed to work with *Minecraft*.

This section covers what is *Minecraft*, its main mechanics, why it was chosen and how it has been previously used in research, as well as other tools we used like *Prismarine* and *Docker*.

2.1 *Minecraft*

Minecraft is an incredibly open sandbox game, that offers players a lot of affordances when it comes to collaborating, whether to gather resources or to put them to use in various creative builds and crafts. It features two main modes. The first is survival, where players have limited health and hunger and start with nothing and must gather resources to survive and eventually grow strong enough to slay the Ender Dragon. The other mode is creative, where players do not have to worry about health and hunger, they can fly around and there is access to an infinite stock of every item in the game. The survival mode poses a great challenge to players, as they must search through the world for resources, manage their items, health, and hunger, and think carefully about how they will spend their time and what they will craft next.

The fact that it is so open-ended, with so many possible actions, constructions, mechanics, and items, means that designers have a lot of options when creating NPCs, without having to add new features to the game. There are also public and private chat features, meaning players and agents can communicate with each other, allowing for dialog.

2.1.1 Uses in Research

There are various *Minecraft* AI competitions currently in the world. They are helpful to learn what has been accomplished in the field using *Minecraft*.

EvoCraft

EvoCraft [4], also known as the *Minecraft* Open-Endedness Challenge is a competition to create algorithms that can generate complex artifacts inside of *Minecraft*. An artifact can be a sculpture, a pattern of blocks that keep changing, or even a procedurally generated Redstone circuit, which to the uninitiated, is a material that essentially acts like electric wiring, allowing players to create simple mechanisms or entire computers inside of the game. The competition is supported by the EvoCraft API, a mod for *Minecraft* that allows the manipulation of blocks in a running *Minecraft* server.

While this competition does not really align with our goals for SocialCraft, it is still very interesting and a great showcase of how *Minecraft* is used in academia. Its nearly infinite set of interactions and combinations is a perfect playground to test algorithms to deal with very complex environments.

Project Malmo

Project Malmo [5] is an AI experimentation platform built by Microsoft on top of *Minecraft*. Its goal is to support the research of artificial intelligence and to provide a way that is not only light on resources but also has a low barrier of entry, to explore the field of AI. It integrates many ideas from the field, such as reinforcement learning, with the end goal of developing agents that can communicate with each other and collaborate with humans. It provides a wide array of tools, that allow for the implementation of a large number of tasks, from a character trying to traverse a treacherous patch of land, to characters that can communicate with humans using natural language.

Microsoft also sponsors competitions using Malmo, such as *MineRL* [6], where contestants train agents using reinforcement learning to obtain a diamond as soon as possible, which requires navigating the complex item hierarchy of *Minecraft*, while also exploring the surrounding environment.

The GDMC Competition

The GDMC Competition [7] is a generative design competition, where participants are tasked with writing an algorithm that will generate a settlement on any *Minecraft* map. These settlements must not only be functional but be aesthetically pleasing and have a narrative to them, so they are comparable to what dedicated humans can accomplish. All this while adapting to the terrain, instead of terraforming the area.

This competition is very much worth studying further, as one of the long-term goals for a framework like ours is to make our agents proactive to the point that they can build and expand their own settlement. While these algorithms build the settlements by placing blocks on their own, their decision-making can be adapted to our agents, so they can build a high-quality village.

2.2 *Prismarine*

PrismarineJS is a *Minecraft*-compatible server (*flying-squid*), bot (*mineflayer*) and Application Program Interface (API) (*minecraft protocol*), all written in Javascript [8]. It has four main projects:

1. **Minecraft data** : Language independent module providing *Minecraft* data for *Minecraft* clients, servers and libraries.

2. **Mineflayer**: Create *Minecraft* bots with a powerful, stable, and high-level JavaScript API.

3. **Flying-squid**: Create *Minecraft* servers with a powerful, stable, and high-level JavaScript API.

4. **Minecraft protocol**: Parse and serialize mine *Minecraft* packets, plus authentication and encryption.

These projects allow programmers to tinker and modify the game, and in our case, create bots whose behavior can be coded. This is how we implemented SocialCraft. Of these, we used the following.

2.2.1 Minecraft data

Minecraft data is a library that contains information about every block (including id, name, hardness, if they're diggable) and other entities of *Minecraft*, like biomes and items. These are all JavaScript Object Notation (JSON) files, that contain all the various properties of each object.

Other *Prismarine* projects use this library as well, to ensure they all can get information about the *Minecraft* world in a consistent fashion.

2.2.2 *Mineflayer*

Mineflayer is an API that allows users to program bots, as in, AI-controlled player characters, for use in *Minecraft*. There are a variety of functions and events that can be used to accomplish this, and bots can be ordered to mine blocks, craft items, go to a certain position, write in the chat box, sleep, etc... All actions that would be expected of a regular player [9].

All the agents' actions in Socialcraft were programmed using the *mineflayer* API, and its modules.

Pathfinder

Mineflayer-pathfinder is a *mineflayer* module that allows users to set goals (such as a specific coordinate, a point adjacent to a block, somewhere in the range of a coordinate, etc..), as well as movement options (if bots can dig, place blocks, sprint, etc...) for each of the bots. Then it will calculate the shortest path to that point and the bots will traverse that path until the goal is met [10].

Collectblock

Mineflayer-collectblock is an expansion of pathfinder, where there is only one goal: to find a block, select the best tool, break it, and collect its dropped item. Unlike pathfinder, you can directly code what happens after the goal is achieved, without having to check if it already has been completed [11].

2.3 *Docker*

Docker is a tool that using OS-level virtualization, allows software to be delivered in packages known as containers [12]. We used Docker to not only deploy the *Minecraft* server for our agents to populate, but also the Socialcraft agents themselves. These containers, using Docker Volume, can persist even after being shut down, allowing us to check their logs and persist their data [13]. There are more details to go into, but those are best left for the implementation section of the document.

Chapter 3

Related Work

3.1 Mods

A mod is an unofficial modification of a game made by people outside of the game's development team. A lot of games, like *Doom* (*id Software*, 1993) encourage modding and create tools specifically for them. In *Doom*'s case, the whole game was even made open source [14]. *Minecraft Java Edition*, the original version and the one we use, unlike its counterparts has no such tools but several mods have arisen because developers have managed to reverse-engineer its Java code [15]. In this section, we will take a look at two mods with goals similar to our objective.

3.1.1 Socialcraft

It makes sense that the first mod we discuss is in fact the previous version of SocialCraft. It worked by letting designers and coders create a configuration file, with a list of identities, agents, and locations, such as houses and workplaces. Agents can have a number of identities assigned to them. An identity dictates how the agent will act at any given time, as in, their available actions. The agent can hold many identities, but only embody one at any given time. Each identity has a salience function, which will determine how likely it is for the agent to pick it at any time, as well as the code required to execute it, which is a series of commands given to an agent for them to perform.

So take the *Lumberjack* identity in listing 3.1 for example. It has a Salience function where if the agent has energy and they are running low on wood, or their favorite wood is nearby, then they are very likely to go and chop wood. This is executed by locating nearby wood blocks, going to and digging them.

```
1 {
2             name: "Lumberjack",
3             variables: {
4                 necessary: ["wood_stock"],
5                 optional : ["favourite_wood"]
6             },
7             salience: [
8                 function() {
```

```
9                         if(this.kb.getValue('energy') > 30){ //how much it wastes to mine a
     block
10                            return (this.kb.getValue("wood_stock") > 15 ? 0 : 0.6)
11                        }
12                        else{
13                            return 0
14                        }
15                    },
16                    function() {
17                        if(this.kb.getValue('energy') > 30){
18                            return this.kb.wasPerceivedVicinity(this.kb.getValue("
     favourite_wood")) * 0.9
19                        }
20                        else{
21                            return 0
22                        }
23                    }
24                ],
25                execute: function () {
26                    const woodBlocks = [35, 36, 37, 38, 39, 40, 46, 41, 42, 43, 44, 45] /*
     wood blocks ids*/
27                    this.locateBlockInArea(woodBlocks, this.get_Forest)
28                    this.digBlock(woodBlocks, this.get_Forest)
29                }
30            }
```

Listing 3.1: Lumberjack identity code

There are other identities like *Eat*, *Sleep*, and *Socialize* dictating when and how the agent will perform those actions. It should be noted that Identities should not be thought of as actions that an agent can perform. They are identities they assume and that dictate their actions. They are not meant to be granular, as in, an identity per action. That implementation of the *Lumberjack* identity is very limited, for example. A full implementation would have a much larger salience function, not only assessing the overall wood stock, but the various different types, how much other NPCs require, what time of the day it is, how durable are their tools right now, etc... Similarly, the execute function would be much larger to account for everything a Lumberjack would do besides chopping wood: fixing their tools, depositing the wood they chop, and much more.

The biggest limitation of the previous iteration of SocialCraft is the authorial effort to craft authentic and rich identities. While this framework makes it much easier to deploy an agent of this depth, it still needs a lot more identities to be coded before it reaches the goals that are set out.

3.1.2 Minecraft Comes Alive

This mod turns villagers into more advanced NPCs, that can be interacted with in a variety of ways not possible on base *Minecraft* [16]. Players can greet them, chat with them, tell them a joke, tell them a story, flirt with them, give them a gift, shake their hand, hug them or kiss them, tell them to follow

you, or to stay in place and trade items with them. Though there are limitations. Only the “greet“ and “tell a joke“ offer dialog options. The others are invisible, as in, the player does not know what they are saying, as it is not shown. Even when telling a joke, the player only gets a description of it and how hard it is to tell. Upon an interaction, a response from the villager is shown on the chat window, and that response varies according to the player’s choice and also the villagers’ feelings towards the player. Gifting villagers, greeting them, telling jokes and a few other actions will raise the hearts of that villagers, a measure of how fond they are of the player. It can be a positive or negative value [16].

Additionally, each villager has a personality, a current mood, and optional traits. A personality is permanent and consistent, and it can be Athletic, Flirty, Sensitive, Stubborn, etc... These have various bonuses, such as running 15% faster, a bonus 25% to all interactions, a 5% chance of losing 35 points each interaction or each interaction loses 15% of its effectiveness, respectively. These personalities can have one of three mentalities: playful, where villagers prefer joking and flirting; serious, where they prefer chatting or telling stories and standard, where they have no real preference [16]. Interactions with the player as well as events around the village (such as a villager dying, for example), alter a villager’s mood. They can be angry, happy, interested, etc... and that will also affect the success of future interactions.

Finally, there are traits, which are optional, so not all villagers feature them. One example is Lactose Intolerance, where the villager can not eat anything with milk, so the player should not gift them dairy food items. Another is Albinism, where the villager lacks pigmentation on their skin. Each villager has genes that determine their physical aspect, down to hemoglobin levels. Genes are determined when villagers have children. The player can also marry a villager and have children with them once their relationship is strong enough. Children can carry out various tasks for the player (such as mining a resource, or chopping wood). After some in-game time, children grow into teenagers and then adults.

This mod already strives for much of what we are planning to do, but it is still very shallow. Interactions are very gamified, where instead of the player having a realistic interaction, they try to pick whichever will raise the number of hearts, for example. The player does not have much control over what they say, NPCs do not have well-defined routines, aside from going to sleep at night, and besides the player’s children (which they can have with villagers in this mod), they can not be asked to complete tasks. They still have the problem default *Minecraft* villagers have of just loitering around instead of working like a society.

3.2 Social Systems Architectures

3.2.1 *Façade*: An Experiment in Building a Fully-Realized Interactive Drama

Façade [17] is an interactive experience where the player, using their own name and gender, plays as a friend of a couple Trip and Grace, during an evening at their apartment where their relationship begins to deteriorate. The player accomplishes this by moving around the apartment, using the mouse to interact with various objects, and writing natural language dialogue that is added to the simulation in real-time, meaning when the player presses Enter to send the line they wrote, it can even interrupt

characters. Trip and Grace are fully voice-acted and animated, with a variety of actions they can perform and emotions to express. The game has multiple interactions and endings, encouraging the player to retry the game multiple times, with them needing about 6 to 7 playthroughs to feel like they have exhausted the possibilities. This is helped by the fact that the experience is only about 20 minutes.

Developed by Michael Mateas and Andrew Stern of Georgia Tech, it was born out of a desire to increase the believability of NPCs in games. As character's visual fidelity continued to increase, their behavior remained simplistic, with players only having a short stock set of phrases to direct at the NPC, a level of simplicity often on par with the player's limited set of physical actions during gameplay (i.e. moving, jumping, crouching, etc...) The lack of a richer array of behaviors and expressivity means games have a harder time handling human relationships. As such, to author agents capable of this type of behavior, it was necessary to use a better approach from finite state machines and scripting languages. These researchers set out with the goal of creating a character-authoring language that not only allows for a designer to have control over the narrative but also the dynamism required to let NPCs not feel overtly scripted. The resulting product was *Façade*, a game where each dialog, gesture, and action of the player alters the game. The narrative can go in wildly different directions but the player cannot perceive set branching points, unlike a Choose Your Own Adventure book, for example.

It achieves this with a framework that allows the creation of structured hierarchies of behaviors, that dictate how such an experience unfolds. However, while the system is procedural because it picks the right behaviors, each individual behavior must still be manually authored, which is very time-consuming. Going further into the system's architecture, in addition to the player, Trip, and Grace, there is an invisible fourth agent, the drama manager, that monitors the game and changes which behaviors and dialogues are available to Trip and Grace. These updates are broken up into story beats, which are a group of behaviors that fit a specific situation, while still offering "a non-trivial simulation space". Beats are created by an author, each with requirements and distinct effects on the plot, and thus the drama manager picks them when they make sense to create an overarching narrative. These characteristics of the beat serve to generate a partial ordering of the beats. Beats change about every minute, from a pool of approximately 200, which can occur in many orders while maintaining narrative coherence. In *Façade*'s *A Behaviour Language* (ABL), each activity is characterized as a goal, each with behaviors that can be used to achieve it. Each behavior is a synchronous or asynchronous sequence of steps, that can also sub-goal its own goals and behaviors, which are kept track of in an active behavior tree. The task of analyzing the current interaction and fitting it into a larger-scale narrative, that also includes all past interactions, falls to the drama manager/beat sequencer. It picks the next beat by checking which can be chosen according to their requirements and then sorting each beat by how they affect the narrative's tension in an Aristotelian story tension value arc, picking a random one according to a specific probability distribution.

Façade is relevant to our project as it presents a simulation that is also very sandbox-like, similarly to *Minecraft*. The player can perform a multitude of actions and say whatever they want and it will have an effect, even if small on the game and how agents behave. Conversely, while we are trying to create a society of NPCs with interpersonal relationships, a web of connections that by itself forms a story, an

overall overarching narrative is not part of our goals, especially with a game as devoid of goals and as infinite as *Minecraft*. The concept of story beats is not useful for us. That, combined with the authorial labor and complexity of creating various behaviors and beats, and the unscalability of the approach, means it was not ideal for SocialCraft.

3.2.2 *Versu* - A Simulationist Storytelling System

Versu [18] is an interactive drama, that is also more of a play than an interactive story, similar to *Façade*. Each player performs their character, even being encouraged to improvise (within the limits of the situation) and each dialogue (or lack thereof) and action is noticed by the agents and will affect their decisions. But while *Façade* revolves around story beats, that dictate what the agents will do, *Versu* is agent-driven, as in, each one makes decisions independently, though there is a drama manager that can influence, not override, their decisions. This true simulation approach was selected to offer more replayability, as something that is not as hard-coded will offer up more story scenarios. Actions are not tied to a specific character and can be performed by all, allowing for a lot more permutations of events. It was also chosen because it gives the player more control over the outcome. “A simulated system has clear rules which the player can learn and internalize.“ the authors of *Versu* say in this paper, as the simulation uses the same models repeatedly. “A nonsimulated system risks being just a series or arbitrary puzzles, in which the player is forced to guess the changing whims of the designer“. To directly address issues from *Façade*, the developers decided to make a text interface aided with some images, for three reasons:

1. Speeding up content production - as modeling, voice acting, and animation are not required
2. Feedback - Text allows to clearly communicate a character's status
3. Better Interface - Rather than an unlimited number of options being converted to one of 30 discourse acts, *Versu* simply gives the player the available options.

Versu is built around two types of objects: agents and their social practices. Social practices are essentially a social situation the agent can partake in (a conversation, a meal, a family, etc...). It coordinates the participating agents, as well as their role in the practice, thus setting the possible actions the agent can take in each situation. While social practices determine the actions available, it is the agent that makes the decision themselves. Multiple practices can exist simultaneously and the total actions available to an agent is the union of all the actions available for each practice. Additionally, some practices are divided into states, which provide different actions in different situations. A social practice is better than a finite-state machine in two main ways. A practice can store arbitrary persistent data instead of just which state it is in. Secondly, in a finite-state machine a state can only transition to another state, while in *Versu*, actions not only change the state but can also add sentences to the database for example, which in turn updates relationships, beliefs/desires available practices.

Since our end goal is to endow the agents with social convention (i.e. do not go to eat in the middle of a conversation, do not ignore a greeting even if the agent is going out to get a resource, etc...), the

concept of social practices dictating what is allowed is a great approach to solve that problem.

These concepts are present in the current implementation of *Socialcraft*, as Identities and Jobs.

3.2.3 *Prom Week*: Designing past the game/story dilemma

Prom Week [19] is a game that revolves around the social lives of 18 characters. In each of the campaigns, each revolving around a different character, the player is given various missions to complete in the week leading up to prom, such as getting a date to attend it with, for example. There are many ways to complete each mission, with the game featuring a new form of gameplay the developers have coined as the *social physics puzzle*. The player could try to befriend a more popular schoolmate, bonding over a shared interest or they could toy with a competitive schoolmate, making an enemy when the player flirts with the schoolmate's romantic interest. Social exchanges are interactions that include multiple characters and modify the social state of their participants. The available exchanges and how they impact the game are managed by the game's AI system *Comme il Faut (CiF)* [20]. *CiF* also determines whether a character will accept/reject an exchange and selects the best outcome from a list of alternatives.

While the game is still heavily authored, with the designers setting the missions and creating a pool of responses for the characters to choose from, *CiF* "enables emergent solutions to each social puzzle" resulting in a game with wildly different stories and responses to player actions. The game was designed so that this open-endedness did not conflict with the goal of telling a coherent story. This is achieved in part by presenting an abstract version of social dynamics. Much like a platformer doesn't simulate accurate physics (characters jump higher, they can change direction in the middle of the air, etc...), *Prom Week* has simplified social dynamics, adjusted for the experience. allowing players flexibility when solving goals, but also consistent and believable characters. This paper describes the architecture as containing the following components:

Relationships: binary, reciprocal, and public connections between characters. The three relationships in *Prom Week* are friends, dating, and enemies.

Social Networks: scalar, non-reciprocal and private feelings from one character toward another. The three networks are buddy, romance, and cool.

Statuses: temporary feelings, either unitary or directional, that are often consequences of social interactions. Some statuses, such as embarrassed, are internal feelings. Other statuses represent social standing, for example, being popular.

Traits: permanent attributes of a character's personality. Most traits are private, such as being competitive, while others are public knowledge, such as being a sex magnet.

Social Fact Database: the social history of interactions between characters. All entries in the social fact database are public knowledge and thus comprise the characters' collective social history.

Cultural Knowledge Base: the objects of the social world, a zeitgeist of popular opinion about each object, and each character's personal relationship to that object, which can

be likes, dislikes, wants or has. For example, Zack may like and want a scientific calculator even though they are generally considered lame.

The first step of the simulation is to form the desires of each character, thus deciding how much they want to play a social game with other characters. This is followed by the player, who chooses the action to perform as well as the target. Each action has an initiator, a responder, and an optional third party. *CiF* chooses how the responder reacts based on the current social context and finally runs a set of “trigger rules“ over the newly generated social state, that account for all the changes that occurred on that timestep. *CiF* largely solves the scalability and authoring time problems that were raised in the previously discussed solutions, as it serves instead not as a framework to support every possible social interaction, but instead focus on the plausible ones, that can be authored in a reasonable timeframe/complexity and that can easily be adjusted by designers. Designers must design objectives and the pool of interactions, but don't need to concern themselves as much with the consequences of each action, the decision-making, and how everything will tie together, as *CiF* is smart to handle all that in a satisfactory manner. It even manages to address the issues of social conventions that were raised with *Versu*. They can be added to the Cultural Knowledge Base, thus dictating a character's behavior according to said conventions.

An issue that has not been mentioned thus far, but is common to all three of the discussed solutions so far, is the lack of routines/proactivity. In *Façade*, *Versu*, and *Prom Week*, since there is an overarching narrative instead of a simulated society, characters do not have routines. They simply react to events, they are maybe even stationary, as in *Prom Week*. But SocialCraft is a simulation of a thriving village, with characters performing their jobs on schedule, and as such routines are implemented.

Socialcraft borrows a lot from Prom Week, like the concept of interactions being one on one, and having to be accepted by two parties, as well as relationships. Though this next work is its biggest inspiration.

3.2.4 A Simple and Method for Evolving and Large Character and Social Networks

Talk of the Town [21] is a text-based game that “simulates a socially oriented American small town from 1839 until 1979.“ And it is a whole town, a large network, as it contains several dozen NPCs, each with mutable but coherent social relationships with each other. This paper presents a generalized approach to creating networks like this, with simple systems that evolve over time from basic social mechanisms. Though the interactions between NPCs are basic they end up creating a rich web of relationships. It operates on one principle: similar characters will bond together, while different characters will antagonize each other. From this, friendships and romances are born. Though nuance is sacrificed, this method allows the generation of a network with hundreds of characters, as opposed to the handful in *Versu* and *Prom Week*. In this system, characters' affinities change according to the social exchanges they partake in with another character. If the other character accepts the exchange, both the character's mutual affinities increase, otherwise, the rejected character will lower their affinity towards

```
while world generation is not over
   advance one timestep
   for each character in the gameworld
      decay charge and spark toward all other characters
      place character at a location
   for each location in the gameworld
      determine which characters at location will interact
   for each dyad of characters that will interact
      for each character in the dyad
         update charge and spark toward other character
```

Figure 3.1: *Talk of the Town* world simulation algorithm

whoever rejected them. It is in essence a lower fidelity version of *CiF*. It is even lower fidelity than that, as the exchanges are even more abstract. They are functions that evolve affinities, instead of heavily scripted behaviors, with a wide array of repercussions and reactions possible. Each character has two core notions towards another character: **Charge**: which is a scalar value representing a friendship/affinity with another character, that increases/decreases in accordance to how compatible/incompatible the agents around them are; and **Spark**: which is like charge, but for romantic feelings. It evolves similarly to Charge.

In order for the simulation to work, basic modeling of time, space, and character personality is necessary. The first two are pretty straightforward, the first is already provided to us in *Minecraft*: the time of day. And the position of the character is given to us in two ways, coordinates, and also by locations in SocialCraft. To recap, SocialCraft has locations set by bounding boxes, which can be houses, workplaces, or social hubs. We can use them to see if two agents are at the same location and also the coordinates to see which are closer together. The final prerequisite is the most complicated, yet most open-ended, as the user can set it however they want, as long as characters have a personality model and a degree of *sociableness*. There must be a way to calculate how likely a character is to engage in a social exchange. It can be a weighted sum of various values, each representing a personality trait from -1.0 to 1.0, for example, though there is a lot of room for different approaches. Additionally, some higher notions are required such as a notion of character proximity (knowing which characters are near others at a certain timestep), friendship compatibility (extroverts will pick more friends, characters of the same gender are more likely to become friends, etc...) as well as how they affect the calculation of their charges, and finally a notion of romantic attraction, which the paper does not elaborate on to save space. Last but not least, a subroutine that lends itself perfectly to fixing our lack of routine problem: placing characters at certain locations on certain timesteps. Characters cannot be placed around randomly. In *Talk of the Town* characters have routines, like going to work, making errands, etc... Since this approach actually requires routines, it solves our previous problem of how to integrate them into one of the solutions. The algorithm itself is described in Figure 3.1.

The paper does mention possible extensions of the system, like different kinds of affinity such as reputation, having more status relationships (for example. *Talk of the Town* has boss-employee and

elder family member-younger family member) as well as more nuanced charge/spark decay, but these might be outside of the current scope of SocialCraft. Still, this is the most relevant solution so far.

The current implementation of Socialcraft features many of these concepts. There are personality traits (though currently, only one), there are friendships (charge), and loves (spark) represented by scalar values, and even the algorithm finds its way in, as characters check others in their location to see if and how they will interact with them. Interactions have to be accepted, and according to the outcome, they can raise/lower the respective friendship/love value. Though, we do not place characters at locations according to the timestep. The agent's actions dictate where the agent will be (though certain actions can only occur at certain times).

Chapter 4

Implementation

4.1 Core Concepts

Socialcraft is comprised of many different classes, all of which need to be explained in order to understand how it works as a whole. These concepts are very intertwined, so I will have to mention some of them before they are fully explained. For the full source code, visit `https://github.com/Catralitos/socialcraft`

4.1.1 Agent

While we can use "agent" to describe the character the user sees move around the *Minecraft* world, and thus, all the code that goes towards accomplishing that is part of the agent, there is an actual Agent.js class.

It stores information about the agent from the deployment process (their friendships, jobs, knowledge, beds, etc...) and also has functions to provide the most salient practices and update its identities, jobs, and knowledge base.

It also stores the corresponding bot. What is the difference between a bot and an agent? A bot is what is deployed by *mineflayer*, and contains info about the player character that is running on the server. The agent is part of Socialcraft. The best way to describe it is, the bot is the body, while the agent is the mind. The bot cannot execute orders without the agent selecting them. The bot can let us know about the world, like its current coordinates, time of day, if it is raining, and other bots in the world, while Socialcraft parses it into instructions. The agent class is the part that makes decisions and stores personal knowledge, but there is the database, which stores general knowledge of practices and jobs, concepts that are also part of Socialcraft. Data for the agent's deployment is stored in JSON and contains all the information we want the agent to know when it spawns, as shown in listing 4.1.

```
1 {
2       "name": "Fran",
3       "jobs": [
4         {"Lumberjack": 0},
```

```
5            {"Miner":  0},
6            {"Patrolman":  0},
7            {"Gatherer":  0},
8            {"Fisherman":  0},
9            {"Farmer":  1}
10         ],
11         "knowledge_base": [
12         ],
13         "personality_traits": [
14           {"Agreeableness": 0.95}
15         ],
16         "friendships": [
17           {"Ash": 5},
18           {"Billie": 5},
19           {"Alex": 5},
20           {"Bobbie":  3},
21
22           {"Casey": 9},
23           {"Charlie": 9},
24           {"Jamie": 9}
25         ],
26         "loves": [
27           {"Ash": 3},
28           {"Billie": 3},
29           {"Alex": 3},
30           {"Bobbie":  1},
31
32           {"Casey": 5},
33           {"Charlie": 5},
34           {"Jamie": 5}
35         ],
36         "bed": "Bed7"
37       }
```

Listing 4.1: An example of an Agent as defined if JSON

Our implementation is significantly different from the previous iteration, as before, the agent also stored all the identities, which in our case, are stored in the Database, and handled the main loop and decision-making, which is now handled in the main.js file, described in section 4.2. It has a much more specific role now.

4.1.2 Database

Before the loop described in 4.2 runs, the agent is created, and then a database is formed that contains all the Jobs, Identities, Practices, and Locations so that this information is accessible to each class that needs it. Besides storing the information, this class features many functions that help access it in an easier fashion.

4.1.3 Location

Locations are defined in Socialcraft as Axis Aligned Bounding Boxes (AABB). Location data is stored in a JSON file like the agents' data and they are comprised of two vertices and a height. Each vertex should share the same y-value, i.e. be on the same plane, to form a cuboid. Both vertices should represent a corner, each diagonally opposed. With those vertices and the height, we can define a three-dimensional box. An agent is in a location if they are in that box.

When locations are added to the database, they are sorted by smallest area first, because when iterating over the list to check which location the agent is in, if there is a location inside another (a house inside a village, a room inside a house, etc...), the smallest, most accurate one gets returned.

There are three types of locations:

1. **Houses** have an extra property: coordinates for the bed where the agent sleeps.

2. **Social Places** have an extra property: social appropriateness. It's a multiplier that is applied to the salience of each social practice if it is happening in that location.

3. **Work Places** have an extra property: can dig and stack. If true, that means agents can place and break blocks inside that location. Unless a location is a workplace where you can dig and stack, it is impossible to do so.

```
1  {
2        "name": "A_House",
3        "bounding_box": {
4          "vertex1": [
5            -6,
6            72,
7            54
8          ],
9          "vertex2": [
10           5,
11           72,
12           34
13         ],
14         "height": 6
15       },
16       "beds": [
17         {
18           "name": "Bed5",
19           "position": [
20             -5,
21             72,
22             51
23           ]
24         },
25         {
26           "name": "Bed6",
```

```
27              "position": [
28                -5,
29                72,
30                47
31              ]
32            },
33            {
34              "name": "Bed7",
35              "position": [
36                -5,
37                72,
38                41
39              ]
40            },
41            {
42              "name": "Bed8",
43              "position": [
44                -5,
45                72,
46                36
47              ]
48            }
49          ]
50        }
```

Listing 4.2: An example of a house as defined in JSON

4.1.4 Practice

A practice is an action that the agent can perform. There is a parent Practice class, from which every individual Practice inherits from. This means there is a .js file for each individual action the agents can do. A practice has eight mandatory functions:

constructor is where the practice is created, given its name, its timeout (more on that in **hasEnded** below), and assigned its bot and agent.

getSalience is the function that returns the salience of the practice, i.e. how likely the agent is to perform this action.

setup is the function where preparations to begin the action take place. If the action is to chat with someone, the agent needs to find out who they will talk to, and where they are. If the action is to break a wood block, they need to search to see if those are available nearby and get their location.

isPossible is the function that checks if the practice can go ahead. If everything went right during setup, the agent is in the necessary state and has the necessary resources, this will return true.

start is the function where the practice sets into motion. Pathfinder goals are set, items are equipped, etc... It is also where the starting time (again, more on that in **hasEnded** below) is set.

update is the function where we can either check on the progress of the practice or perform something to further advance it. Unlike the previous functions, which are called only once, update is called periodically. As such, it is used to check if the agent has already reached a certain location, or to trigger events that can not happen on **start**, that must start later.

hasEnded is the function that checks if the practice has ended. This can usually happen in two ways: the practice has ended naturally and achieved what it set out to do, or, the agent has been performing the practice for longer than the timeout value set in the **constructor** (hence why we needed a starting time in **start**. This may happen because the agent got stuck, or the practice took longer than expected.

exit is the function called when the practice ends. It usually is used to set attributes like the starting time or the pathfinder goal back to *null*.

You can see the *ChopWood* practice in Listing 4.3. In **getSalience**, the agent checks if they are a Lumberjack and if they are in work hours. In **setup**, the agent searches for a wood block inside their workplace, and stores it. The practice is only possible if a block was found. In **start**, the agent selects the best tool and calls **getBlock**, which using *mineflayer-collectblock*, sets a pathfinder goal for the agent to get to the block and break it, thus discarding the need for an **update** function to periodically check if they have reached the block. After **getBlock** is done, the agent stores that they finished digging, one of the conditions to end the practice (the other being the timeout), and on exit, it resets its properties.

```
1  const woodIds = [17, 162]
2
3  class ChopWood extends Practice {
4
5      _targetWoodBlock;
6      _finishedDigging;
7
8      constructor(bot, agent, timeout = 20) {
9          super(bot, "ChopWood", agent, timeout);
10         this._targetWoodBlock = null;
11         this._centralPoint = null;
12         this._finishedDigging = false;
13     }
14
15     getSalience(context) {
16         return this._agent._current_job._name === "Lumberjack"
17         && this._agent._current_job.onTheJob(this._agent._bot.time.timeOfDay) &&
18         this._agent._current_job._location === context._location ? 2 : Number.
       NEGATIVE_INFINITY;
19     }
20
21     setup(context) {
22         this._centralPoint = this._agent._current_job._location.getCentralPoint()
23         let radius = this._agent._current_job._location.getRadius()
24         let blocks = this._bot.findBlocks({
```

```
25                  point:  this._centralPoint,
26                  matching: block => {
27                      return block.name.endsWith("_wood") || woodIds.includes(block.type)
28                  },
29                  maxDistance: radius,
30                  count: 50
31              }
32          )
33          if (blocks.length > 0) {
34              let index = Number.parseInt(this._agent._bed[3]) >= 5 ? 0 : Math.floor(Math.
    random() * blocks.length)
35              this._targetWoodBlock = this._bot.blockAt(blocks[index])
36          }
37      }
38
39      start() {
40          super.start()
41          let bestTool = this._bot.pathfinder.bestHarvestTool(this._targetWoodBlock)
42          if (bestTool !== null) {
43              this._bot.equip(bestTool, "hand").then(() => "Equipped " + bestTool.
    displayName);
44          }
45          this.getBlock()
46      }
47
48      getBlock() {
49          begin(this._targetWoodBlock, this._bot).then(r => {
50              this._finishedDigging = true
51          })
52
53          async function begin(targetWoodBlock, bot) {
54              if (targetWoodBlock) {
55                  try {
56                      await bot.collectBlock.collect(targetWoodBlock)
57                  } catch (err) {
58                      console.log(err) // Handle errors, if any
59                  }
60              }
61          }
62      }
63
64      isPossible() {
65          return this._targetWoodBlock
66      }
67
68      hasEnded() {
69          return super.hasEnded() || this._finishedDigging;
70      }
71
```

```
72      exit() {
73          super.exit()
74          this._agent.incrementItemInKnowledgeBase("wood_stock")
75          this._targetWoodBlock = null;
76          this._finishedDigging = false;
77          this._bot.pathfinder.setGoal(null);
78      }
79  }
```

Listing 4.3: ChopWood.js

4.1.5 Social Practices

Social Practices extend Practices further and are actions that involve dialog between agents. They have the exact same functions as the Practice class (in fact, while every social practice inherits from SocialPractice.js, that very class inherits from Practice.js), but with one new addition. The **accepts** function, which will be explained shortly. Though it retains the same functions, they are used in slightly different fashion.

A social practice behaves like Figure 4.1. Agent A initiates it, and Agent B reads it in chat. If agent B accepts the social exchange, they will reply, which will be detected by Agent A, and end the practice. *Greet* is a practice that begins a social interaction. Each agent can have a current social partner, who they are chatting with. *Greet* assigns it, beginning the interaction, while *Goodbye* does the opposite, terminating the interaction and setting each agent's social partners to *null*. Let us explain further by using the *Greet* social practice as an example.

Look at listing 4.4. When we create the practice, we also set an event: when someone chats ”Hello” followed by the agent's name, the agent will see if they accept the social interaction, and if they do not already have a social partner, will assign it, as well as mark the agent as socializing. The thing about the code is that only here, with the assistance of booleans (a type of variable that can only have two values: true or false), can the agent know if they are Agent A or Agent B. That is why we keep track of a $_chatted$ boolean, to know if the agent already has spoken. If they have not chatted yet, that means they are Agent B detecting Agent A's message, and thus, they must reply (which we also store as a boolean). Otherwise, this means Agent A has already greeted Agent B, they are detecting Agent B's reply, and that the practice is done.

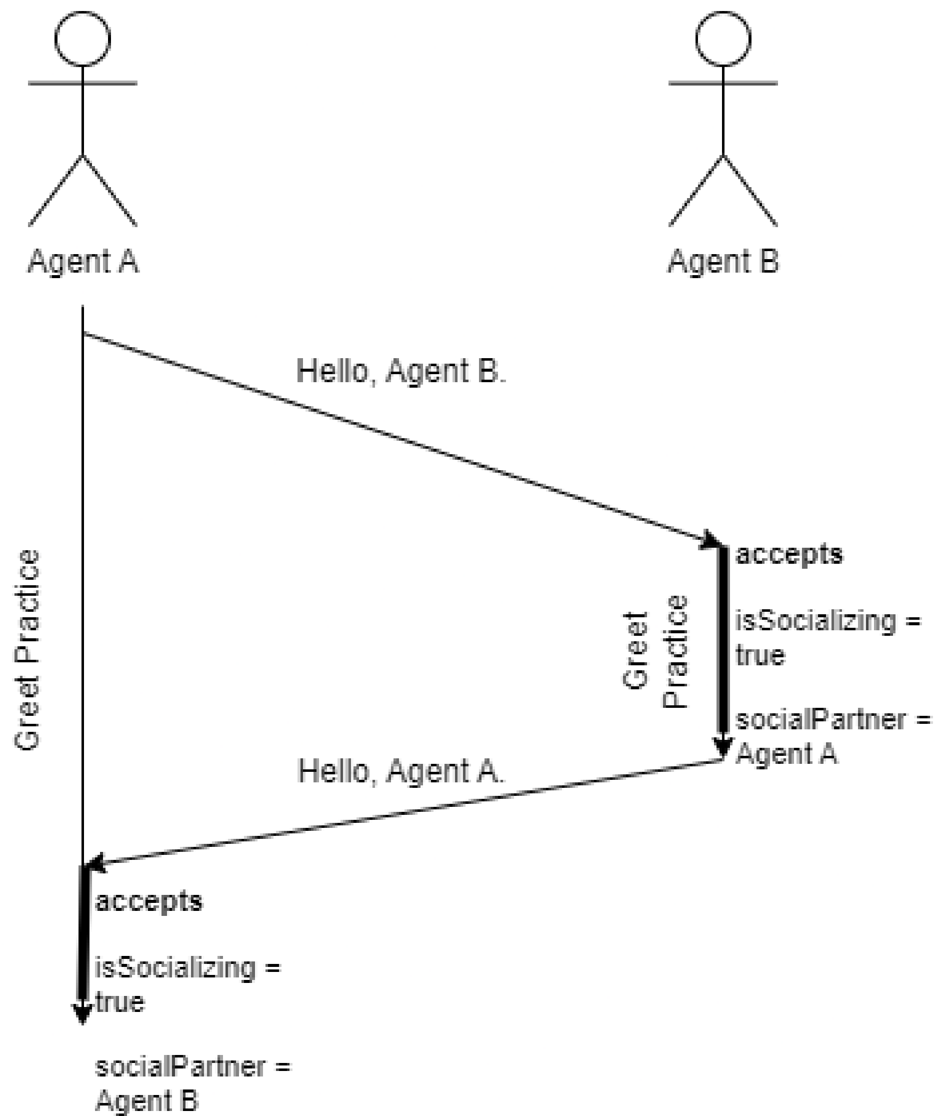

Figure 4.1: A diagram exemplifying how the Greet practice works. Agent A decides to greet Agent B. When Agent B hears it, he runs his *accepts* function and agrees to it, marking that he is socializing and his social partner is Agent A. He replies, and Agent A then undergoes the same process.

```
constructor(bot, agent, timeout = 20) {
        super(bot, "Greet", agent, timeout);
        this._bot.on("chat", (username, message) => {
            if (message === 'Hello, ' + this._bot.username) {
                if (this.accepts(username) && !this._agent._socializing) {

                    this._agent._socializing = true;

                    function getBotByUsername(username, bot) {
                        let players = Object.values(bot.players);
                        for (let i = 0; i < players.length; i++) {
                            let botAux = players[i]
                            if (botAux.username === username && botAux.entity != null) {
                                return botAux
                            }
                        }
                        return null
                    }

                    this._currentTarget = getBotByUsername(username, this._bot)

                    //if I haven't said hello, I must reply
                    if (!this._chatted) {
                        this._mustReply = true
                    }
                    //if I have said hello, then I got a reply, and I'm done
                    else {
                        this._done = true
                    }
                }
            }
        })
    }
```

Listing 4.4: Greet.js constructor

Next, there is the **getSalience** function, shown in Listing 4.5 It begins with an *if* statement, to check if the player is already socializing. If they are, that means they have already received a message. Thus we need to use the $_mustReply$ boolean to check if it is Agent A or Agent B. Let me remind you, that for the agent to be already socializing, that means the chat event triggered and they already accepted the interaction. This means it is a question of whether they must reply or not. At that point, we consider nothing more important than replying (or in the case of ignoring the other agent, unimportant). That is why we use positive and negative infinities as saliences. Positive if they must reply, negative otherwise. If the agent is not socializing, that means they have not chatted, and have not yet decided to chat. So the agent iterates over the others surrounding them, sees who they are more likely to chat with (a mix of the agent's own agreeableness and their friendship value towards others), and returns that. The agent

also remembers their current target, i.e. the person who they are more likely to talk to.

```
1  getSalience(context) {
2          //if it accepted, then it must reply thus a high salience
3          if (this._agent._socializing) {
4              return this._mustReply ? Number.POSITIVE_INFINITY : Number.NEGATIVE_INFINITY
5          } else {
6              //else see if there is someone around you want to greet
7              let highestSalience = -1;
8
9              for (let i = 0; i < context._listOfSurroundingPeople.length; i++) {
10                 let otherBot = context._listOfSurroundingPeople[i]
11                 let currentSalience = this._agent._personality_traits["Agreeableness"] *
       (this._agent._friendships[otherBot.username])
12                 if (currentSalience > highestSalience) {
13                     highestSalience = currentSalience
14                     this._currentTarget = otherBot
15                 }
16             }
17             return highestSalience;
18         }
19     }
```

Listing 4.5: Greet.js getSalience

Moving on to **accepts**, in listing 4.6 the function which is exclusive to Social Practices. If two agents accept to partake in social interaction, their mutual friendships will go up. Thus there needs to be a check for it, that if true, increases their mutual affinity, or else, decreases it. The check for greet is identical to the salience and returns true if that value is bigger than 2.

```
1  accepts(username) {
2          let accepted = (this._agent._personality_traits["Agreeableness"] * this._agent.
       _friendships[username]) > 2
3          if (accepted) {
4              this._agent._friendships[username] = clamp(this._agent._friendships[username]
        + 0.2, 0, 10)
5          } else {
6              this._agent._friendships[username] = clamp(this._agent._friendships[username]
        - 0.2, 0, 10)
7              this._done = true
8          }
9          return accepted
10     }
```

Listing 4.6: Greet.js accepts

socialPractice.js includes in its **start** code to set a pathfinder goal towards the social partner, and **update** has code that not only updates the target (because the partner may move) but also sets a $_nearTarget$ boolean to true, which each individual social practice may use to trigger their dialog. After

an agent speaks it is marked as having chatted, though only in the case where the agent had to reply (thus, they were Agent B), is the practice marked as done and ready to exit.

4.1.6 Jobs

A job is a vocation that the agent can have. Right now, it is represented as a scalar value, so in essence, they have saliences just like practices do. Though, in the current implementation of SocialCraft, those values are constant and do not change during the course of the simulation. This is so future iterations can have the job the agents take on in the future be more mutable.

A job essentially describes three things:

1. Where the agent works
2. When the agent works
3. What the agent can do when they work

Each job, besides their name and affinity, has a workplace, where the agent is supposed to work, and a list of Time Blocks.

Time Blocks

A Time Block is an object with start and end times, both in hours (so 9 is 9 o'clock, and 9.5 is 9:30 o'clock in *Minecraft* time), and a list of practices. Meaning, while the agent has a certain job, during the time intervals set by a time block, they can only execute these practices. Thus we can prevent the player from eating outside of lunch hours, or trying to mine ore while they are a Lumberjack, or even partaking in excessive socialization.

Job Definitions

There is a file in the project, *jobDefinition.js*, which contains the definition for each job: their name, location, and time blocks. In listing 4.7, you can see how the Lumberjack job is defined.

```
1  const JOB_DEFINITIONS = [
2      {
3          "Name": "Lumberjack",
4          "Location": "Forest",
5          "TimeBlocks": [
6              {
7                  "StartTime": 9,
8                  "EndTime": 13,
9                  "Practices": ["GoToWork", "ChopWood", "Compliment", "Goodbye", "Greet", "
       Insult", "WeatherTalk"]
10             },
11             {
12                 "StartTime": 14,
13                 "EndTime": 17,
```

```
14                      "Practices": ["GoToWork", "ChopWood", "Compliment", "Goodbye", "Greet", "
        Insult", "WeatherTalk"]
15                  }
16              ]
17          }
18  ]
```

Listing 4.7: Definition of Lumberjack in jobDefinitions.js

4.1.7 Context

Context includes all that is surrounding information relevant to the agents' decision-making, that they already don't know themselves, but they gather from their surroundings. Their location, the weather, who is in the same location as them, and in future iterations, even more data.

```
1       _location;
2       _listOfSurroundingPeople;
3       _weather;
4       _isWorking;
5       _database;
```

Listing 4.8: Attributes of context.js. The database is included as it is the only way to pass it on to the practices for a consultation currently.

4.1.8 Identities

An identity is almost like a persona that an agent can embody. Depending on the context of the situation, they may choose to adopt one or multiple of them. If they are surrounded by people they dislike, they may adopt the "Enemy" identity, while if they are working, they will adopt the "Working" identity.

Each identity checks if they are valid for that context, as in if the agent should embody them. An identity, boiled down to its simplest definition, is a set of rules that define how the salience of each action is affected. Let us look at listing 4.9, for the salience rules of the identity "Friend". When the agent is surrounded by friends, he will be friendly. As such, practices such as "Greet" and "Complement" get a bigger multiplier, so they become more salient. While practices like "Insult" get such a low multiplier, they are practically guaranteed not to occur.

```
1   this._salienceRules = {
2               "Greet": 1.2,
3               "Compliment": 1.75,
4               "Insult": 0.1,
5               "Goodbye": 0.9,
6               "AvoidPeople": 0.3
7           }
```

Listing 4.9: Salience Rules in the constructor of friend.js

When an agent goes to check the salience of their available practices, the multiplier for each of the currently valid identities is applied. So if they have a salience for a practice of 5, but two separate active identities, one with a 0.1 multiplier and another with a 1.5 multiplier for that practice, both will be applied for an end result of 0.75, as shown in equation 4.1

These identities are much different than those from the previous iteration of Socialcraft, as those were essentially practices, that only had a salience and execution function (with no setup, start, update, execution, or exit). Also, some of those identities corresponded to jobs, for example. The concept there was muddled, so we split it into three different ones: practices, jobs, and identities, to make the framework more complete and versatile, allowing for more actions to be coded for the agents.

$$5 * 0.1 * 1.5 = 0.75 \tag{4.1}$$

Salience after two separate identities rules for that practice have been applied.

4.2 Agents' Main Loop

Each agent, throughout their lifetime, executes the same core loop, similar to the algorithm in Talk of The Town, as shown in Figure 3.1. The loop is as such:

There are a few more bells and whistles in the implementation, for logging purposes, and to ensure that if an agent finishes a practice too soon, they don't immediately pick another, but this is how the selection of the agent's actions is handled. This function is in the *main.js*, the default class that the bot runs when deployed.

This is vastly different from the previous iteration of Socialcraft, where the "loop" was a periodically called *mineflayer* event, that simply picked the most salient "identity" at each time. Now not only do we gather much more to affect the actions the agents take, but also we check to see if the actions are done or possible before executing them or giving up on them.

4.3 Deployment

4.3.1 Bots

Socialcraft's deployment was created by Diogo Rato, with a few later tweaks by myself. In essence, for each agent, a Docker container is created. By building a Dockerfile (show in listing 4.10), we can install on the container the Node Package Manager (npm) packages required for the agent to run (like *mineflayer* and its sub-projects), and copy the Socialcraft handler.

The handler is a class that makes the bridge between the system that deploys the agents and the containers. The agents are deployed using a Python script, deploy.py, that reads the info of the scenario from ¡the SON files (one for the agents, another for the locations), connects to the specified Minecraft server and deploys bots with certain environment variables, i.e., the information that was on the JSON

Algorithm 1 Socialcraft Agent Main Loop

```
1: function ASYNCBASICAGENTLOOP(handler, agent, normalMove, digAndStacMove)
2:     ongoingPractice ← null
3:     while true do
4:         if bot is not sleeping then
5:             check which job agent will pick
6:             gather context of surroundings
7:             gather location from context
8:             set movement type according to location
9:             activate valid identities
10:            if ongoingPractice ≠ null then
11:                if ongoingPractice is no longer possible OR ongoingPractice has ended then
12:                    exit ongoingPractice
13:                    ongoingPractice ← null
14:                else
15:                    update ongoingPractice
16:                end if
17:            else
18:                get availablePractices
19:                get mostSalientPractice from availablePractices
20:                if mostSalientPractice ≠ null then
21:                    ongoingPractice ← mostSalientPractice
22:                    setup ongoingPractice
23:                    if ongoingPractice is possible then
24:                        start ongoingPractice
25:                    else
26:                        ongoingPractice ← null
27:                    end if
28:                end if
29:            end if
30:        end if
31:    end while
32: end function
```

files. The handler not only spawns the bots on the server, it allows users to access the information on the servers inside the containers.

```
from nikolaik/python-nodejs:python3.10-nodejs16-alpine
LABEL socialcraft_agent=''

# Add Mineflayer modules
WORKDIR /agent
RUN npm install js-logger mineflayer mineflayer-pathfinder mineflayer-statemachine
    minecraft-data mineflayer-collectblock vec3 dotenv
RUN npm install

COPY socialcraft_handler.js/ ./socialcraft_handler.js

# Run Agent
EXPOSE 3000
CMD ["node", "main.js"]
```

Listing 4.10: Socialcraft Dockerfile

4.3.2 Server

To deploy a server, we used Docker Compose. That means, we created a *YAML Ain't Markup Language* (yml/yaml) file, shown in listing 4.11 that contains information about the server we want to create.

```
version: "3.8"

services:
  minecraft-server:
    image: itzg/minecraft-server:latest
    container_name: minecraft-server
    ports:
      - 25565:25565
    environment:
      SERVER_NAME: "Socialcraft-Minecraft-Server"
      EULA: "TRUE"
      VERSION: "1.12"
      MODE: "survival"
      ONLINE_MODE: "false"
      OPS: "${OPS_PLAYERS}"
      MAX_PLAYERS: 9
      SPAWN_ANIMALS: "true"
      SPAWN_MONSTERS: "false"
      SPAWN_NPCS: "false"
      PVP: "false"
      GENERATE_STRUCTURES: "false"
      MAX_WORLD_SIZE: 100
      OVERRIDE_SERVER_PROPERTIES: "true"
```

```
24          SPAWN_PROTECTION: "0"
25          WORLD: /worlds/test-world.zip
26          #RCON_CMDS_ON_CONNECT: |-
27          #  /give @a baked_potato 64
28
29        restart: "no"
30        user: "${UID}:${GID}"
31        volumes:
32          - ./worlds:/worlds:ro
33          - ./data:/data
34        logging:
35          driver: "local"
36          options:
37            max-size: "1m"
```

Listing 4.11: docker-compose.yml

For starters, we have to specify what we wanted to run on the containers. In our case, we used *itzg's Docker Minecraft* server, which allows you to host a *Minecraft* server on a *Docker* container [22]. Then, we specify the port, and *Minecraft* servers default to port 25565. We also used a previous version of *Minecraft*, 1.12, to ease the load on our machine as an older version takes fewer resources.

Following that, there are environment variables, that allow us to control the details of the world we are spawning. For example, in our case, we prevented the spawning of other NPCs, as well as monsters that could kill the agents. We set a max world size, as there was no use having a gigantic world, etc...

Finally, in our case, we specified the volume. Usually, *Docker* creates a virtual volume in Windows, which was what we used. It is in essence a virtual storage unit that hosts the containers, and cannot normally be accessed. The other alternative is bind mounting, where the container's files are stored directly on the host machine. To quote directly from *Docker's* documentation:

"Volumes have several advantages over bind mounts:

- Volumes are easier to back up or migrate than bind mounts.
- You can manage volumes using Docker CLI commands or the Docker API.
- Volumes work on both Linux and Windows containers.
- Volumes can be more safely shared among multiple containers.
- Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
- New volumes can have their content pre-populated by a container.
- Volumes on Docker Desktop have much higher performance than bind mounts from Mac and Windows hosts.

In addition, volumes are often a better choice than persisting data in a container's writable layer, because a volume does not increase the size of the containers using it, and the volume's contents exist outside the lifecycle of a given container." [23]

While we used volumes for the bots' containers, we used bind mounting for the server. This is

because, we wanted to make it so each time a test subject went into the server, the map was identical. Volumes allow for persistence, which means even as we shut down the server, when we restarted it, the map's changes would still be there. What we needed was a way to initiate the server identically each time. So, we bind mounted it, setting the files to a folder called ”data” in our project repository, as well as creating a ”world” folder, to store worlds we wanted to deploy. We extracted the map, as a zip file, from the data folder, and placed it in the world folder. Finally, we added an environment variable, **WORLD**, that pointed to the zip file. Thus, this means, each time we deploy the server, as long as we delete the container and the data folder, it clones that world directly.

4.3.3 File Structure

In our project repository, there are several folders. Let us cover what each folder is for in more detail.

- **examples** are the various implementations of agents' AI that can be done. Socialcraft at its core is a deployment tool for bots. The logic that governs them can come in many forms.
    - **phase-one** is the implementation accomplished for this thesis, and the one described in this chapter. We called it phase-one because it will be expanded upon later by our successors. It contains the deployment script.
        - **blueprint** is the folder containing the Dockerfile and main.js scripts required for the agents to work. It has two subfolders with the bulk of the thesis' code.
            - **config** contains implementations of the core concepts as well as files unique to each scenario.
            - **database** contains every implemented identity, practice, and social practice (there is a subfolder per category).
            - **scenarios** contains a scenario per subfolder. A scenario consists of an Agents.json and Location.json pair, defining the agents and the world around them.
            - **logic** contains key classes like Agent, Database, Context, etc... Modifying any of these files causes major changes.
- **images** contains the various possible implementations of the Socialcraft handler and Dockerfile, according to language.
    - **base_nodejs_image** contains socialcraft_handler.js and the Dockerfile (as mentioned in 4.3.1. This is a Javascript implementation.
- **node_modules** contains all installed npm packages for the project to work.
- **socialcraft** contains Python scripts called on by the deployment script to successfully launch the bots.
- **venv** contains all installed Python packages for the project to work.
- **worlds** contains zip files with each world we might want to deploy into a server.

Chapter 5

Evaluation

In this section, we explain how we tried to evaluate how successful our framework was. We set various goals and parameters according to aspects we wanted to test, like agent believability and framework flexibility.

5.1 Evaluation Goals

5.1.1 Believability of the agents

Our believability metrics came from *Metrics for Character Believability in Interactive Narrative* [24], which studied how *Disney* brings characters to life through narrative and animation and took those principles to enumerate various dimensions of believability that can be used to gauge how believable and AI agent is. They are as follows:

- Behaviour coherence
- Change with experience
- Awareness
- Behaviour understandability
- Personality
- Emotional Expressiveness
- Sociability
- Visual Impact
- Predictability

By measuring through factors, designers can identify where their agents lack. If they lack in personality, they could try to give them more unique traits or interactions, while if they lack visual impact, there need to be more or more elaborate animations.

So as the paper suggests, there are Likert scales on the questionnaires, asking subjects how much they agree with the fact that the agents present each of these characteristics.

Emotional expressiveness will not be tested, because that requires a more in-depth analysis of a single agent by the subject, and requires us to ask questions like ”What was agent X feeling at this particular time?” with multiple choice answers. Given how Socialcraft is not really scripted so the same things happen every time.

5.1.2 Sense of society

In addition, we are including questions about if the agents felt like they were part of a bigger society and if they had noticeable daily routines. Though the test in [24] was designed for a single agent, we are adapting it to many, because our goal is to deploy a society of agents, a large number, with little authoring effort, and these two traits are part of our goals.

5.1.3 Flexibility of the framework

We also wanted to evaluate Socialcraft’s flexibility, i.e. how many different kinds of agents it can generate, even with shared building blocks (practices, jobs, and identities), just by altering individual agents’ properties, like their personality traits and affinities to other agents. This is why we decided to have two different societies that provide contrast with each other. If the differences were noticeable to subjects, that would indicate that goal was achieved.

5.2 Scenario

5.2.1 Test Environment

For the test, we built a Minecraft map, featuring two villages: Village A and Village B, which when the subject spawns in, are to their left and right, respectively. Village A features the agents:

- Alex (Lumberjack)
- Ash (Miner)
- Billie (Gatherer)
- Bobbie (Fisherman)

The only buildings in the village are four nearly identical houses, one for each agent. The agents of this village were designed to have low agreeableness and to have low friendship values with the other agents, thus, making them overall less sociable. They conduct their activities solo, and rarely partake in social practices. This village can be seen in figure 5.1.

Village B on the other hand is much more sociable. It features the agents:

- Casey (Lumberjack)

- Charlie (Miner)
- Fran (Farmer)
- Jamie (Guard)

The village features a farm and three buildings: a shared house with four rooms, one for each agent, a canteen, and a bar. These users are not only more agreeable and friendly with each other, but they also conduct a lot more activities together like their meals, and at night, before sleeping, they hang out together at the bar. This village can be seen in figure 5.2.

The map featured other key locations: a forest for the lumberjacks to chop trees on, a mine for the miners to mine ore, and a wharf for the fisherman to fish. Fran works on the Village B farm, harvesting carrots, Billie gathers plants from all around the world, and Jamie patrols the Village B walls.

Figure 5.1: Village A

For each test, we hosted a Minecraft server and the bots on our machine, while asking remote subjects to connect using their own version of *Minecraft Java Edition* 1.12. Tests were monitored over video calls where users would share their screens. For the test, users were tasked with following Casey around from 9 a.m. to 7 p.m. (Minecraft Time, just over 8 real minutes), which encompasses a whole work day, and post-work socializing. Casey was chosen as she works with Alex, providing a direct contrast between agents from both villages. This is depicted in figure 5.3

The goal was for the test subject to use this opportunity to observe a routine with various activities and contrast it with one more plane like Alex's, though we must stress, and we did to each subject, the goal was to observe all of the agents, as the questionnaire was about all of them.

Each time the subject connected to the server, we used its command line to give them some baked potatoes (in case their character got hungry or lost health from a fall) and set the time to 8 a.m. After this, we began the spawning of the agents, so when 9 a.m. began, the subjects were already in place to follow Casey. The routine is rough as follows:

Figure 5.2: Village B

Figure 5.3: Casey working with Alex. Note that Casey has greeted Alex.

1. Leave Village B and go to the forest to work with Alex from 9 a.m. to 1 p.m.

2. Ditch Alex, and go to the canteen to have lunch with the rest of Village B's residents until 2 p.m.

3. Return to the forest and resume working with Alex until 5 p.m.

4. Ditch Alex, and go to the canteen to have dinner with the rest of Village B's residents until 6 p.m.

5. Go to the bar and socialize with the other residents of Village B until 7 p.m.

6. Go to sleep

A number of different social interactions can occur in between. After the subject saw Casey go to sleep, we shut down all *Docker* containers, and output their logs to text files using the Windows Powershell, keeping logs of all containers for each of the test subjects. The agents' logs contained information about their location, coordinates, practice, identities, and decision-making, at frequent time intervals.

5.3 Problems

The tests did not turn out as expected and could have gone better. This is because they were plagued with a myriad of problems.

5.3.1 Scope Problems

Not only did implementing the framework take longer than expected, but also, the scenarios, i.e. the content to showcase it (going back to 4.3.3, what would be in the config folder) was not scoped properly and had a few unforeseen problems, primarily with the *mineflayer* API.

In an attempt to reach the goals discussed in 5.1.3, we ended up with a wide variety of bots that did not work satisfactorily instead of a few that worked really well. We could not refine pathfinding and dialog, for example. We also could not solve the problem of equipping items. When an agent requires a pickaxe, if it is not in their inventory, *mineflayer* can not give it to them. It needs to be given through commands on the server command line or attained in-game, which is impractical. We did attempt to run these commands for each agent when they connected to the server, through environment variables in the docker-compose.yml file, but we had to remove them because they caused the server to crash.

Jobs are very simplistic. Lumberjacks only chop wood, they do not deposit it, and they do not craft with it. Same for miners. The original plan was to have carpenters and cooks and to have the other agents give them resources, but when we ran out of time for that, instead of scaling back on the number of agents, and potentially dropping the concept of two different villages, we scaled back on the jobs instead, which led to jobs like Guard and Gatherer, which are not nearly as interesting.

Also, while social practices worked exactly as intended when there were only two or three agents in a server when there were eight, dialogues became more erratic and disjointed, as agents coped with the large number of agents to choose from, and also sometimes two agents picked the same agent for social interaction, causing problems.

So we ended up planning the test in a way where subjects could focus on the bots that work better, leaving those which were not working as well in the background. This was not to be misleading, as the bots' problems were still obvious even when subjects did not follow them around, but given the limited time we have a subject for, it was imperative to get feedback on the bots that did work because in that case, we could not immediately gauge how successful (or not) they were.

5.3.2 Technical Problems

On top of that, there were technical problems. One of the main disadvantages of using *Docker* on Windows, is that you cannot directly control how many resources are allocated to it. This means, sometimes, bots just ran out of memory and crashed. Part of the reason we could not do a more in-depth test with the other villagers, like the miners, is that they kept crashing early on. Meaning there are parts of the map/scenario that no tester got around to seeing because we could not send them there, so that was wasted work.

Additionally, we were plagued with *mineflayer* bugs: empty beds saying they were not empty; when the bot tried to sleep we got an error saying the bot was not sleeping when attempting to use a bed, which is the point; tried to eat food that did not exist; events not calling in time; events calling over each other and causing errors, etc... A lot of errors deep in the *mineflayer* API, we could not fix or find workarounds in time. *mineflayer-pathfinder* also usually just did not work properly on occasion, with agents trying to walk straight lines into a block, instead of circumventing it, or running into solid walls.

Also sometimes when bots began deployment, the process would timeout before all were deployed, meaning we would have to reset the time on the server, and try to deploy again, sometimes up to 4 times before the bots deployed. This seems to be an issue with *Docker* we could not fix.

5.4 Test Procedure

All the tests were conducted remotely with people who owned their own copies of *Minecraft Java Edition*. We asked the subjects to open the questionnaire which you can find in Appendix A (A). The first page explained what Socialcraft was and some demographic questions about their level of experience with Minecraft.

Following that, the questionnaire explained how they were to proceed with the test on the server, and we also explained and assured them of what to do. They proceeded to follow Casey around for the day.

Once that was finished, while we stored the logs of the *Docker* containers, the users answered filled out a series of Likert scales, regarding the metrics discussed in sections 5.1.1 and 5.1.2.

After that, we asked them to write about the differences between the agents in both villages, and finally gave them the option to write some feedback, if they wanted to.

In the end, we thanked them for their participation.

Chapter 6

Results

In this section, we will go question by question through the survey we asked our subjects to fill out, and draw conclusions. The questionnaire can be found in Appendix A (A). There were a total of twenty tests performed.

6.1 *Minecraft Experience*

These were demographic questions designed to gauge the subject's level of experience with *Minecraft*.

6.1.1 Overall Experience

1. What is your level of experience with Minecraft?
20 respostas

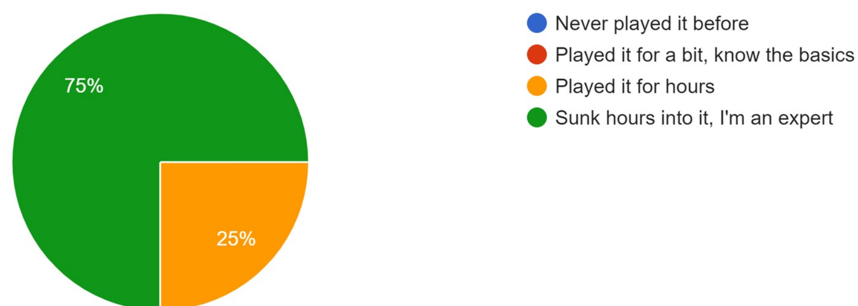

Figure 6.1: The distribution for the level of experience with Minecraft

As seen in figure 6.1, most had a lot of experience with *Minecraft*'s mechanics, and none were new to the game. That meant they were better able to understand what the agents needed to do to achieve their tasks.

6.1.2 Multiplayer Experience

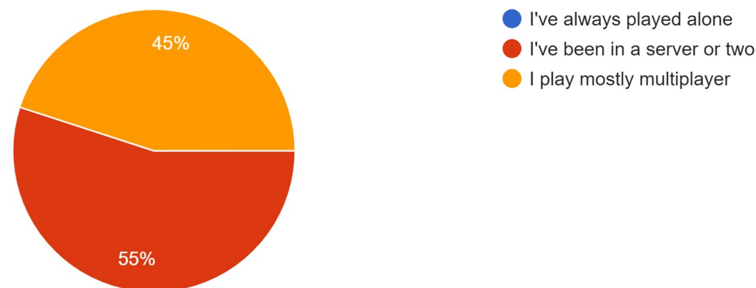

Figure 6.2: The distribution for the level of experience with Minecraft multiplayer

As seen in figure 6.2, all users had at least some experience in playing with other players, meaning they have taken part in a Minecraft society of their own and have used its communication tools.

6.2 Believability

Following the test, subjects were asked to fill out the rest of the form. They were presented with several statements, and given Likert scales, that went from 1 - Strongly Disagree to 5 - Strongly Agree. We will consider [1,2] as not agreeing, [3] as being ambivalent, and [4,5] as agreeing.

For each of the survey's questions, we will calculate the average, and sum the values to a possible score of thirty (which is five times six, as for reasons explained in section 6.2.8, which will be clear). This value will give us an idea of how successful our agents were.

6.2.1 Awareness

$$\frac{1*0+2*4+3*4+6*4+6*5}{20} = 3.70 \tag{6.1}$$

Average value of awareness

When asked if they believed the agents were aware of their surroundings, 60% agreed, as seen in Figure 6.3, with an average value of 3.70, as seen in formula 6.1. Though the results are promising, increasing the information gathered in Context would be a big step towards improving these results. Making it so a practice could be interrupted, could also curb some of the cases where it seems an agent is ignoring their surroundings.

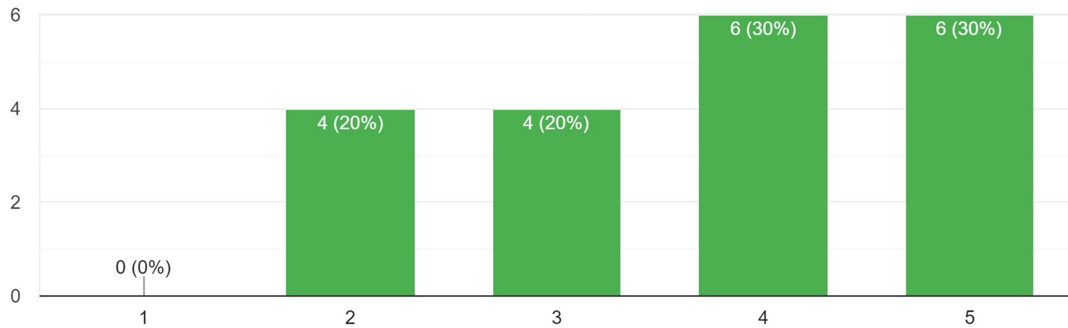

Figure 6.3: The distribution of agent awareness

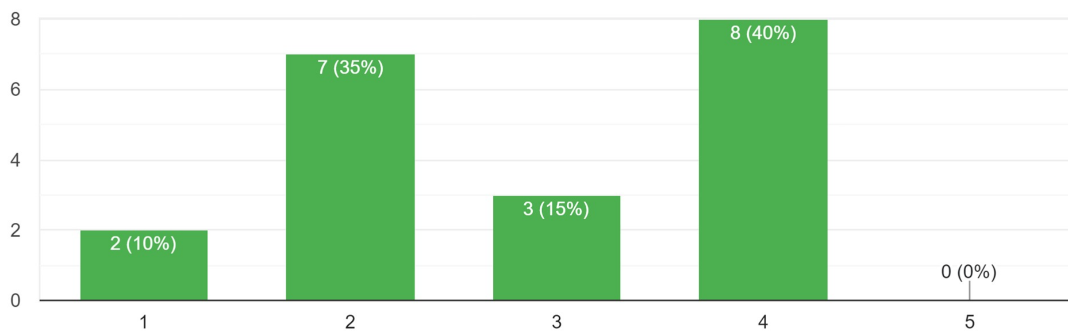

Figure 6.4: The distribution of agent understandability

$$\frac{1*2+2*7+3*3+8*4+0*5}{20} = 2.85 \tag{6.2}$$

Average value of understandability

6.2.2 Behaviours Understandability

When asked if the agents' behavior was easy to understand, the results were very polarizing. While 40% of people agreed, 45% disagreed, as seen in figure 6.4. We can assume that part of the reason for the polarizing results, was the inconsistent nature of the tests. As mentioned before, pathfinding sometimes did not work, and also frequently agents spent far too long running back and forth to socialize instead of performing their jobs. To fix this, it is not that necessary to look at the decision-making, but at how actions are carried out. If they happen in a more natural and expected manner and have their saliences tweaked, it can be improved. Given the results, the average ended up being only 2.85, as

seen in the formula 6.2, which was the lowest of all the attributes.

6.2.3 Personality

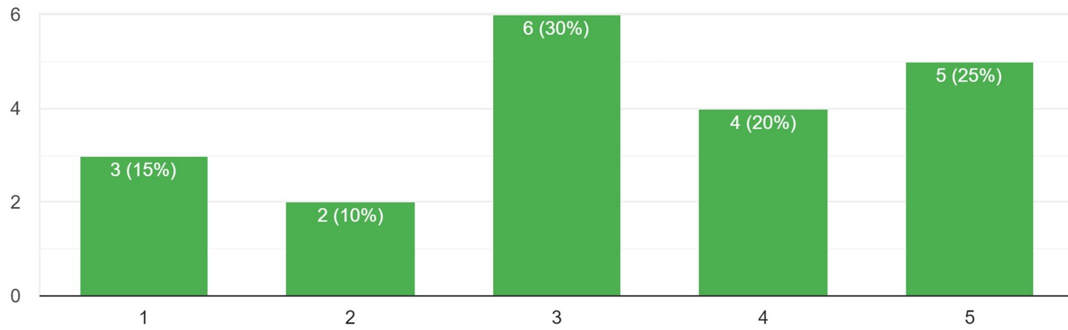

Figure 6.5: The distribution of agent personality

$$\frac{1*3+2*2+3*6+4*4+5*5}{20} = 3.30 \tag{6.3}$$

Average value of personality

When asked if the agent's exhibited unique personality traits, the results were also very polarizing. The most common response was a 3, ambivalence, with 30% of subjects answering that. Though overall it was more positive as 45% agreed and only 25% disagreed, as seen in figure 6.5. This makes sense, as in this implementation, instead of a set of personality traits, like in Talk of the Town, there was only one: agreeableness. And given how it only fluctuates from 0 to 1, if salience functions were not properly made to cope with how nuanced the results can be, boiling down to hundredths, then different behaviors according to that trait might not emerge or seem noticeable. More traits and bumping the values from [0,1] to [0,10] might fix this.

Also, each social practice only had a dialogue option, and it did not sound very natural (though there still needs to be a debate over if we want our NPCs to talk more like players, casually, or more formally like characters). Adding more dialog options for the same practices, which may occur multiple times, will alleviate the repetition, and making them dependent on the personality will help personalities to emerge.

Overall, the average value of the answers was 3.30 as seen in formula 6.3.

6.2.4 Visual Impact

When asked if the agents' actions drew their attention, i.e., had visual impact, most people seemed to agree, 70% in fact, as shown in figure 6.6. This is good to know, as it is one of the few aspects

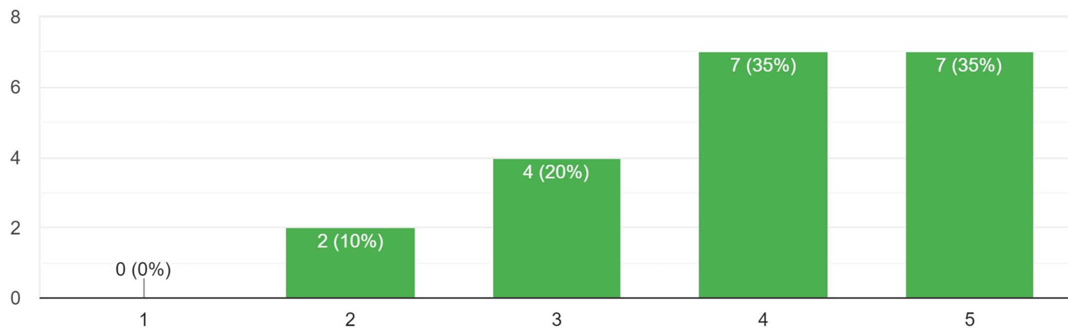

Figure 6.6: The distribution of agent visual impact

$$\frac{1*0+2*1+3*4+4*7+5*7}{20} = 3.85 \tag{6.4}$$

Average value of visual impact

we have little to no control over. We cannot insert new animations, facial expressions, or effects into Minecraft, at least not with the tools currently used. Though the high 3.85 average, as seen in formula 6.4, can probably be attributed to the pathfinding. Agents not only sprint around very fast in a way that draws attention but also parkour around, stacking blocks and jumping walls instead of circumventing them. While fixing pathfinding to have more natural movement may have an adverse effect on this, programming more practices should keep the balance just right.

6.2.5 Predictability

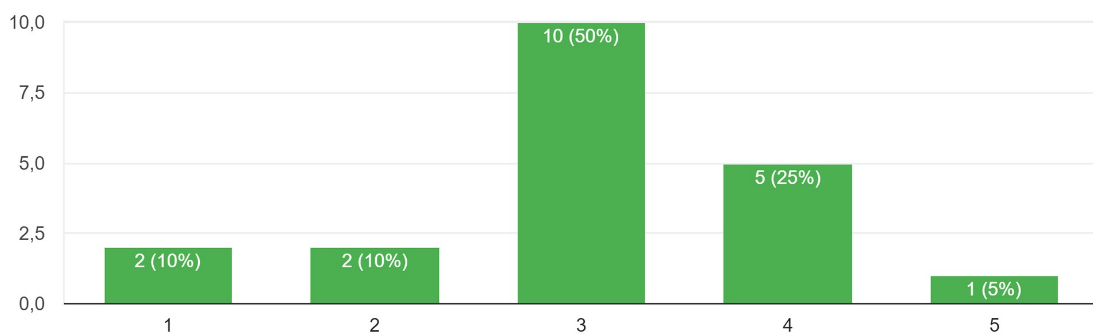

Figure 6.7: The distribution of agent predictability

$$\frac{1*2+2*2+3*10+4*5+5*1}{20} = 3.05 \tag{6.5}$$

Average value of predictability

When it came to predictability, half the subjects answered 3, ambivalence, as shown in figure 6.7. This is not necessarily bad, as mentioned in [24], if agents are too predictable, it is hard for them to be believable. This is a problem older NPCs had, with a stock set of interactions, the type of problem we are attempting to fix. Conversely, too much unpredictability is borderline incoherence, so the fact the average turned out near the middle, 3.05, as seen in formula 6.5 is not a cause for alarm. We feel it should be a bit higher, but improving the practices should help fix that.

6.2.6 Behaviour Coherence

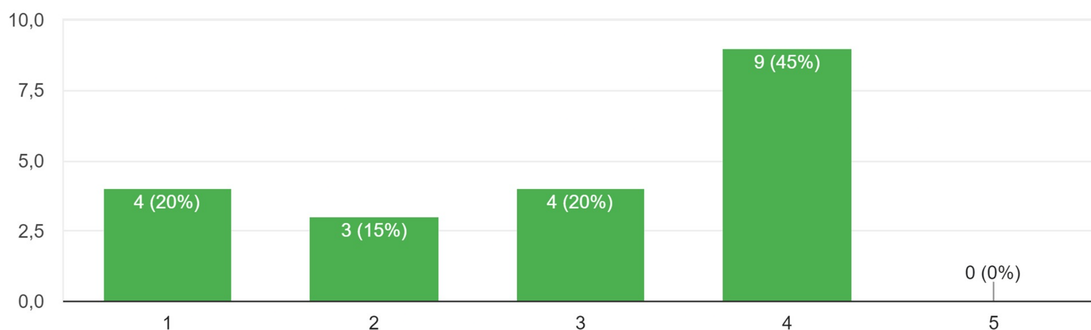

Figure 6.8: The distribution of agent coherence

$$\frac{1*4+2*3+3*4+4*9+5*0}{20} = 2.90 \tag{6.6}$$

Average value of coherence

Speaking of coherence, it was a whole other story. It scored the second lowest average out of each category, 2.90, as seen in formula 6.6. Looking at figure 6.8, we can see that 45% of users did agree they were coherent. But none strongly agreed. The fact the other 55% did not agree, brought down the average a lot. This once again comes down to the practices: they need to be redone to be more complete and the salience functions need to be refined, as to avoid the agent repeating too many actions. Weak pathfinding and the lack of items (both tools and food) for the agents to equip when they were required also surely contributed.

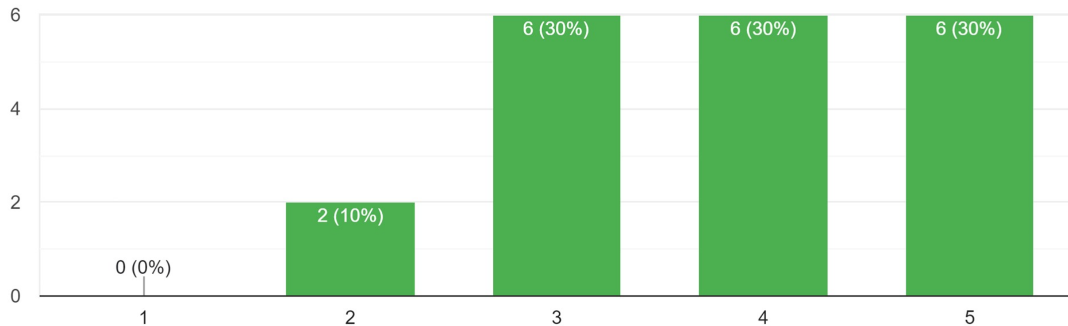

Figure 6.9: The distribution of agent sociability

$$\frac{1*0+2*2+3*6+4*6+5*6}{20} = 3.80 \tag{6.7}$$

Average value of sociability

6.2.7 Sociability

When it came to sociability, only 20% of people did not agree the agents interacted socially with each other, as seen in figure 6.9, yielding a 3.80 average, as seen in formula 6.7. Agents often picked social interactions with each other but given only 30% of people strongly agreed, and from our observations, the quality of the interactions could be higher. Agents often just greeted each other and then said goodbye, repeating this loop, not having a conversation between the beginning and end of the social interaction, but also trying to restart a conversation as soon as it ended. To fix this, not only do the salience values of social interactions need to be redone, but also, there must be some sort of ”cooldown” period between conversations between the same two agents.

6.2.8 Final Tally

$$3.70 + 2.85 + 3.30 + 3.85 + 2.90 + 3.80 = 20.40 \tag{6.8}$$

Sum of the average values for each question

In the end, our agents got a score of 20.40 out of a possible 30 points, as seen in formula 6.8. Though we should note, since predictability is an attribute that we do not want to be maxed out, as explained in 6.2.5, we did not include it.

$$\frac{20.40*100}{30} = 68 \tag{6.9}$$

Rule of three to convert the score to 100 point scale

Using a rule of three to convert that to a more readable 100-point scale, as shown in formula 6.9 we get a final score of 68 out of 100, which while not terrible, is also not great. As we pointed out in each category there is a lot of room for improvement.

6.3 Sense of Society

In this section, we will discuss the metrics not associated with believability, though also tested with Likert scales. Since these were not designed to be used as numerical metrics, like those in section 6.2, they are kept separately here.

6.3.1 Daily Routines

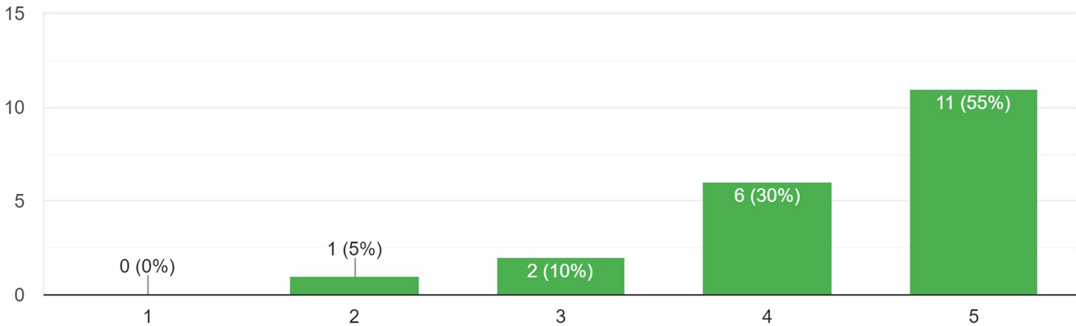

Figure 6.10: The distribution of agent adherence to daily routines

$$\frac{1*0+2*1+3*2+4*6+5*11}{20}=4.35 \tag{6.10}$$

Average value of adherence to daily routines

Adherence to daily routines scored very highly, yielding a 4.35 average, as seen in formula 6.10. Not only did 85% of people agree they adhered to daily routines, 55% strongly agreed so, as seen in figure 6.10. The implementation of jobs certainly contributed to this. Though this is slightly problematic. A lot of the aspects of daily routines such as lunchtime and sleeping time were less dictated by factors like hunger or tiredness, but by using positive and negative infinite salience values to force agents to perform certain actions at certain times. And while we want them to always eat at mealtime, a little flexibility as to when in that period would enhance the experience, and make it seem more varied. There could also be legitimate reasons for an agent to skip a meal, he got stuck in a conversation or was helping out someone else, for example.

6.3.2 Society

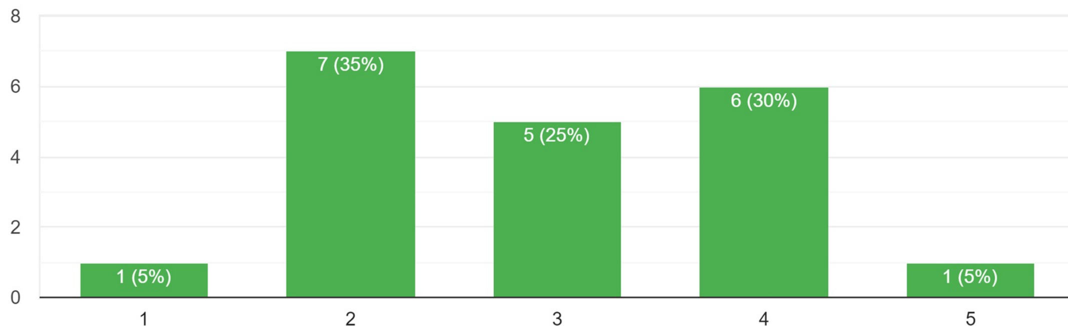

Figure 6.11: The distribution of agent belongingness in society

$$\frac{1*1+2*7+3*5+4*6+5*1}{20} = 3.15 \tag{6.11}$$

Average value of belongingness in society

Finally, there was the feeling of the agent belonging in society, with very mixed results, as only 10% of people had strong feelings about it, as seen in figure 6.11. Still, the most predominant answer was to disagree, taking up 35% of the results. We attribute this to the fact that besides social interactions, which as discussed in 6.2.7, were flawed, agents do not interact or cooperate. Their jobs and tasks do not intertwine, they do not have a shared common goal to work towards, and no non-social practice requires more than one agent. If we address these problems, we can raise the average value from its current 3.15, as seen in formula 6.11, which is very low.

6.4 Social Differences

Following that, we asked our subjects to write about which differences they noticed between the two villages' agents. Instead of giving them a set of answers to choose from we decided text would be better, as we did not even want to hint at what the differences would be and influence them in any way. We just wanted to glean what they observed.

Unfortunately, due to how the test was structured, with its many problems as mentioned in 5.3, a very common feeling was that subjects did not notice differences, or more accurately, did not feel exposed to village A agents enough to notice any differences. Eight subjects mentioned this.

Of the rest, a lot of them seemed to notice that village A's agents were a lot less talkative than village B's, which is what was intended. This was pointed out by seven of our subjects.

Other things that were pointed out were:

- Village B agents were a lot less productive than Village A agents, focusing on socializing rather than work
- Village B agents went to the canteen to have their meals, while Village A agents remained in their workplace
- Village B agents would often interrupt work and go to different workplaces to socialize, unlike Village A agents
- Village A and Village B agents focused their interactions on agents of the same village and did not interact with agents from the opposite village

This is promising, as even with a limited set of practices, we were able to instill into both our sets of agents fundamental differences that set them apart and were noticeable. Still, if future iterations want to test the versatility of the framework, it is key to prepare a better test and make the bots function better overall. It would also be interesting to try to apply some subtler, more nuanced differences to see if subjects would pick up on them.

6.5 Assorted Feedback

Finally, there was an optional opportunity for the subjects to write feedback and suggestions for future iterations of Socialcraft. Here are some of the criticisms, that we already have not mentioned in this document and how we plan to address them:

- The bots run really fast - it is really easy to fix, all we have to do is turn off sprinting in the pathfinder movement options.
- Alex, the lumberjack, did not break a tree until the end or pick up the blocks - While the picking up blocks may be a bit harder to fix, as *mineflayer* does not seem to have functions for that, the randomness of which blocks Alex is easy to explain: they are in fact random. This was done so two lumberjacks running the same practice would not pick the same block (as if it were not random, they would always pick the closes block to the center of their workspace, the forest). But given that Casey almost always chose to try to socialize with Alex instead, all that happened is that Alex was running around breaking random blocks. The practices need to be updated so lumberjacks chop down a whole tree, and try not to break the same tree together.
- Agents often stood in the same exact position - we need to use different pathfinding goals, as well as ways to check if the goal has been met, so when we send agents to a location, like the canteen, they stand around on different spots

Chapter 7

Conclusions

At the start of this thesis, our goal was to create a tool to deploy robust agents into Minecraft, by expanding on the previous iteration of Socialcraft. When designing the framework, we tried our best to not only keep it simple, but also easy for designers to set up and adjust, and also to add new content to it. The overall results were mixed. The framework itself was well conceived and developed, leaving a good setting-off point for further expansion. And we are confident the framework itself was fine because given how the scenario content went, we would not have achieved even these results without it. We had daily routines, we had the basis for social interactions, and we had a context that affected decisions, identities, and much more. Sadly there were a lot of setbacks. Due to burnout and poor time management, the project kept getting delayed. With an over-ambitious scope, it led to not as much getting done as we wanted, and that hurt the final product, and thus, the test results.

Having that said, the results were not wholly negative, and there were a lot of aspects like awareness, visual impact, predictability, daily routines, and sociability, which are already in good places. We can not fully measure our contribution to authoring efforts yet, as there is still a lot of work to be done. But also, a lot of work has been done, and it will be easier for our successors to keep going.

7.1 Achievements

- Deployment of the server and agents - using *Docker*, we have provided a streamlined and easy way to achieve the deployment

- Development of a framework for agent behavior - the core concepts and code for the agents' behavior, as mentioned in section 4.1, were well conceived and implemented

- Creation of a small society - while there were problems, Village B's villagers did end up feeling and behaving like a society

7.2 Limitations

- Lack of finished content - it is clear that when creating the practices, we did not use *mineflayer* to its fullest potential, and they were incomplete
- Difficulty in creating map JSON - going around finding the corners and height of each building/location to insert them into JSON is very impractical compared to the rest of the deployment process
- *Mineflayer* and *Docker* technical problems - the bugs and memory shortages mentioned in 5.3 need to be fixed, as the current solution is not stable enough.
- Poor testing - it is very hard to have a test subject observe a whole society of agents, let alone two. And it is clear we did not find the correct solution.
- Lack of evaluation of authoring efforts reduction - this is still a key question we could not answer, partly because we were the only ones who deployed a society and frankly, it is not something we can ask just anybody to do. Testing needs to be done with people who have experience in the matter, but not with Socialcraft. It should happen in the next iteration, as such tests are as important as the ones conducted so far.

7.3 Future Work

In the future there are several things that could be done:

- Reduce repeated code in practices by better-using hierarchies - for example, right now there are two nearly identical practices: Eat and EatSocial. In the former, the agent eats on the spot, and in the latter, they go to a social place, in our case the canteen, first. They are very similar and reuse a lot of code, which could be solved by using hierarchies even for implementations of the Practice class. This might help fix the problem of multiple agents trying to do the same thing like chopping wood, as there can be variations on the task.
- Turn Socialcraft into a Minecraft mod - while the tests were run with a server deployed by us, who also ran the deployment, making it much easier so a broader audience can access it would be a benefit. Using Java code, we could theoretically do a Minecraft mod, like Minecraft Comes Alive, which is much easier to install and runs the proper command line arguments in the background to launch agents onto a server.
- Fix map building - We built our map by hand, it would have been nice to have a Docker-compatible tool to facilitate the process.
- Have a simulation manager - One of the problems of the current implementation is that each agent exists wholly independent from the other. A *mineflayer* bot can get access information of another bot, but not the Socialcraft agent. This means, for example, a Lumberjack has no way of knowing if an agent around them is a Lumberjack. This could be resolved by adding it to the knowledge base

of the agents, but for a large number of agents that is a lot of work to do manually. We could have the bots whisper (a private chat not displayed to all) questions to each other to get information. But ideally, a manager of the whole simulation each bot can access would be best. This would also be a good place to store the Database, instead of each agent keeping a copy.

- Conduct tests on Authoring Efforts reduction - as explained in section 7.2.

Bibliography

[1] J. Valcarcel. How one man invented the console adventure game. Wired, Mar. 2013. URL https://www.wired.com/2015/03/warren-robinett-adventure/. Acessed October 2022.

[2] G. E. Team. 15 most influential games of all time. Gamespot, . URL https://web.archive.org/web/20100515053341/http://www.gamespot.com/gamespot/features/video/15influential/p9_01.html. Acessed October 2022.

[3] T. Kogod. 11 ways dungeons dragons influenced video games. TheGamer, Sept. 2020. URL https://www.thegamer.com/ways-dungeons-dragons-influenced-video-games/. Acessed October 2022.

[4] D. Grbic, R. B. Palm, E. Najarro, C. Glanois, and S. Risi. Evocraft: A new challenge for open-endedness. 2012.

[5] M. Johnson, K. Hofmann, T. Hutton, D. Bignell, and K. Hofmann. The malmo platform for artificial intelligence experimentation. In *25th International Joint Conference on Artificial Intelligence (IJCAI-16)*, July 2016. URL https://www.microsoft.com/en-us/research/publication/malmo-platform-artificial-intelligence-experimentation/.

[6] W. H. Guss, C. Codel, K. Hofmann, B. Houghton, N. S. Kuno, S. Milani, S. Mohanty, D. P. Liebana, R. Salakhutdinov, N. Topin, M. Veloso, and P. Wang. The minerl competition on sample efficient reinforcement learning using human priors. In *Thirty-third Conference on Neural Information Processing Systems (NeurIPS) Competition track*, December 2019. URL https://www.microsoft.com/en-us/research/publication/the-minerl-competition-on-sample-efficient-reinforcement-learning-using-human-priors/.

[7] C. Salge, M. C. Green, R. Canaan, and J. Togelius. Generative design in minecraft (gdmc). Technical report, 2018.

[8] P. Team. Prismarinejs. GitHub, . URL https://prismarine.js.org/. Acessed on October 2022.

[9] M. Team. Mineflayer. PrismarineJS, . URL https://mineflayer.prismarine.js.org/#/. Acessed October 2022.

[10] M. pathfinder Team. Mineflayer-pathfinder. GitHub, Oct. 2022. URL https://github.com/PrismarineJS/mineflayer-pathfinder#readme. Acessed October 2022.

[11] M. collectBlock Team. mineflayer-collectblock. GitHub, May 2022. URL https://github.com/PrismarineJS/mineflayer-collectblock. Acessed October 2022.

[12] D. D. Team. Overview. Docker Docks, . URL https://docs.docker.com/get-started/. Acessed October 2022.

[13] D. D. Team. Persist the db. Docker Docs, . URL https://docs.docker.com/get-started/05_persisting_data/. Acessed October 2022.

[14] J. Carmack. Readme.txt. GitHub, Dec. 1997. URL https://github.com/id-Software/DOOM. Acessed October 2022.

[15] E. Maiberg. Why gamers are worried about 'minecraft: Windows 10 edition'. Vice, July 2015. URL https://www.vice.com/en/article/53984z/why-gamers-are-worried-about-minecraft-windows-10-edition. Acessed Jan 2022.

[16] M. C. A. Team. How to play mca. Radix Shock Mods, June 2014. URL https://web.archive.org/web/20210510135611/http://www.radix-shock.com/mca--how-to-play.html. Acessed via Wayback Machine on January 2022. (Capture is from May 10, 2021, 13:56:11).

[17] M. Mateas and A. Stern. Façade: An experiment in building a fully-realized interactive drama. 2003.

[18] R. Evans and E. Short. Versu -a simulationist storytelling system. IEEE, 2014.

[19] J. McCoy, M. Treanor, B. Samuel, A. A. Reed, M. Mateas, and N. Wardrip-Fruin. Prom week: Designing past the game/story dilemma. Technical report, 2013.

[20] J. McCoy, M. Treanor, B. Samuel, A. A. Reed, M. Mateas, and N. Wardrip-Fruin. Social story worlds with comme il faut. IEEE, 2014.

[21] J. Ryan, M. Mateas, and N. Wadrip-Fuin. A simple and method for evolving and large character and social networks. 2016.

[22] itzg. Readme. GitHub, Oct. 2022. URL https://github.com/itzg/docker-minecraft-server. Acessed October 2022.

[23] D. D. Team. Use volumes. Docker Docs, . URL https://docs.docker.com/storage/volumes/. Acessed October 2022.

[24] P. G. A. P. C. M. A. Jhala. Metrics for character believability in interactive narrative. Technical report, 2013.

Appendix A

Questionnaire

This appendix contains the questionnaire used for user testing.

Socialcraft Believability Test Form

Hello, my name is Carlos Marques, I am completing my master's degree at Instituto Superior Técnico and my thesis is *Socialcraft*.
Socialcraft is a framework that allows the deployment of agents with coded behavior into a *Minecraft* server. To put it in a simpler way, in the server I will ask you to join, there will be player characters that actually aren't humans at all. They are agents whose behavior I have coded, to make the world more sociable. You're here to help me gauge just how believable these agents are, and how they could be improved in the future.

ATENTION: Your participation is anonymous. There won't be any registry of your identification.

*Obrigatório

1. 1. What is your level of experience with *Minecraft*? *

Marcar apenas uma oval.

- ◯ Never played it before
- ◯ Played it for a bit, know the basics
- ◯ Played it for hours
- ◯ Sunk hours into it, I'm an expert

2. 2. What is your level of experience with *Minecraft* multiplayer? *

Marcar apenas uma oval.

- ◯ I've always played alone
- ◯ I've been in a server or two
- ◯ I play mostly multiplayer

The test

In this server, there will be two villages: Village A and Village B.

Your task for this test is to follow a bot from Village B. In doing so, we hope you observe each village's differences, as well as ALL the agent's behaviors, not just the one you're following.

Step 1. Connect to the server

In the *Minecraft* title screen, head to Multiplayer, then pick Add Server. In the **Server Name** box, type in *Socialcraft* and in the **Server Address** box type the following:

94.62.236.32:25565

Press the Done button, and now, the *Socialcraft* server should appear in your server list, which you only need to click to join.

Step 2. Follow Casey till nightfall

Casey is from Village B, and works as a Lumberjack, with Alex from Village A. Use this opportunity to observe everyone's behaviours.

Believability

For each statement about the game's agents, please choose how much you agree with them.

3. 1. The agents perceive the world around them *

Marcar apenas uma oval.

	1	2	3	4	5	
Totally Disagree	◯	◯	◯	◯	◯	Totally Agree

4. 2. It is easy to understand what the agents are thinking about *

Marcar apenas uma oval.

	1	2	3	4	5	
Totally Disagree	◯	◯	◯	◯	◯	Totally Agree

5. 3. The agents have distinct and identifiable personality traits *

Marcar apenas uma oval.

	1	2	3	4	5	
Totally Disagree	◯	◯	◯	◯	◯	Totally Agree

6. 4. The agents' behaviors draw my attention *

Marcar apenas uma oval.

	1	2	3	4	5	
Totally Disagree	◯	◯	◯	◯	◯	Totally Agree

7. 5. The agents' behavior is predictable *

Marcar apenas uma oval.

	1	2	3	4	5	
Totally Disagree	◯	◯	◯	◯	◯	Totally Agree

8. 6. The agents' behavior is coherent *

Marcar apenas uma oval.

	1	2	3	4	5	
Totally Disagree	◯	◯	◯	◯	◯	Totally Agree

9. 7. The agents' have noticeable daily routines *

Marcar apenas uma oval.

	1	2	3	4	5	
Totally Disagree	⬭	⬭	⬭	⬭	⬭	Totally Agree

10. 8 . The agents interact socially with each other *

Marcar apenas uma oval.

	1	2	3	4	5	
Totally Disagree	⬭	⬭	⬭	⬭	⬭	Totally Agree

11. 9. The agents feel part of a larger society, rather than existing on their own *

Marcar apenas uma oval.

	1	2	3	4	5	
Totally Disagree	⬭	⬭	⬭	⬭	⬭	Totally Agree

Social Differences

As I mentioned, Village A and Village B have their share of differences.

12. **What social differences between both societies did you notice?** *

Focus on social behaviors, how they talk, how they commune with each other, etc...

Further feedback

The *Socialcraft* project will continue even after I'm done with it, not only to improve the agent's behaviours, but to get it into the hands of more people. So if you have feedback, please leave it here.

13. **Do you have any feedback you'd like us to know about?**

Thank you for participating in this form!

Este conteúdo não foi criado nem aprovado pela Google.

Google Formulários