Social Agents in Minecraft

Carlos Marques[1]

[1] *Instituto Superior Técnico, Lisbon, Portugal*

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

I. Introduction

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 role-play 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.

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.

II. 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*.

a. 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 vari-

Contact data: Carlos Marques, carlos.a.marques@tecnico.ulisboa.com

ous 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.

b. Prismarine

PrismarineJS is a *Minecraft*-compatible server (*flying-squid*), bot (*mineflayer*) and Application Program Interface (API) (*minecraft protocol*), all written in Javascript [4]. 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.

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 Java Script 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. 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 [5].

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 [6].

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 [7].

c. Docker

Docker is a tool that using OS-level virtualization, allows software to be delivered in packages known as containers [8]. 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 [9]. There are more details to go into, but those are best left for the implementation section of the document.

III. Related Work

a. 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 [10]. *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 [11]. In this section we will take a look at two mods with goals similar to our objective.

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 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 {
```

```
            name: "Lumberjack",
            variables: {
                necessary: ["wood_stock"],
                optional : ["favourite_wood"]
            },
            salience: [
                function() {
                    if(this.kb.getValue('energy') > 30){ //how much it wastes to mine a block
                        return (this.kb.getValue("wood_stock") > 15 ? 0 : 0.6)
                    }
                    else{
                        return 0
                    }
                },
                function() {
                    if(this.kb.getValue('energy') > 30){
                        return this.kb.wasPerceivedVicinity(this.kb.getValue("favourite_wood")) * 0.9
                    }
                    else{
                        return 0
                    }
                }
            ],
            execute: function () {
                const woodBlocks = [35, 36, 37, 38, 39, 40, 46, 41, 42, 43, 44, 45] /*wood blocks ids*/
                this.locateBlockInArea(woodBlocks, this.get_Forest)
                this.digBlock(woodBlocks, this.get_Forest)
            }
        }
```

Listing 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.

b. A Simple and Method for Evolving and Large Character and Social Networks

Talk of the Town [12] 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 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 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

```
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
```

Fig. 1: *Talk of the Town* world simulation algorithm

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. 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 specific actions can only occur at certain times).

IV. Implementation

a. 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.

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.

2. Database

Before the loop described in d 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.

b. 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, as 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. 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*.

2. 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 as such: 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 2. 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.

```
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 2: Greet.js constructor

Next, there is the **getSalience** function, shown in Listing 3 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.

```
getSalience(context) {
        //if it accepted, then it must reply thus
    a high salience
        if (this._agent._socializing) {
```

```
            return this._mustReply ? Number.POSITIVE_INFINITY : Number.NEGATIVE_INFINITY
        } else {
            //else see if there is someone around you want to greet
            let highestSalience = -1;

            for (let i = 0; i < context._listOfSurroundingPeople.length; i++) {
                let otherBot = context._listOfSurroundingPeople[i]
                let currentSalience = this._agent._personality_traits["Agreeableness"] * (this._agent._friendships[otherBot.username])
                if (currentSalience > highestSalience) {
                    highestSalience = currentSalience
                    this._currentTarget = otherBot
                }
            }
            return highestSalience;
        }
    }
```

Listing 3: Greet.js getSalience

Moving on to **accepts**, in listing 4 the function which is exclusive to Social Practices. If two agents accept to partake in a 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.

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

Listing 4: 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.

3. 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.

4. 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.

5. Job Definitions

There is a file in the project, *jobDefinition.js*, which contains the definition for each job: their name, location and time blocks.

6. 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 surroundings. Their location, the weather, who is in the same location as them, and in future iterations, even more data.

c. 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 5, 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.

```
this._salienceRules = {
            "Greet": 1.2,
            "Compliment": 1.75,
            "Insult": 0.1,
            "Goodbye": 0.9,
            "AvoidPeople": 0.3
        }
```

Listing 5: 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.

d. Agents' Main Loop

Each agent, throughout their lifetime, executes the same core loop, similarly to the algorithm in Talk of The Town, as shown in Figure 1. The loop is as such:

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
```

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 selection of the agents' actions is handled. This function is in the *main.js*v, the default class that the bot runs when deployed.

e. Deployment

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

1. Server

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

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 [13]. 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 less resources.

Following that, there are environment variables, that allow us to control details of the world we are spawning. For example, in our case, we prevented the spawning of other NPCs, as well as monsters which 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 which 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. [14]

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.

V. Evaluation

a. Evaluation Goals

1. Believability of the agents

Our believability metrics came from *Metrics for Character Believability in Interactive Narrative* [15], 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.

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 [15] 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.

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.

b. Scenario

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.

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. 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.

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.

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 roughly as follows:

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.

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 B (**??**). 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 1 and 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.

VI. Results

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. **NOTE:** for brevity we will only include results related to agent believability.

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

a. Predictability

$$\frac{1*2+2*2+3*10+4*5+5*1}{20} = 3.05 \qquad (1)$$

When it came to predictability, half the subjects answered 3, ambivalence. This is not necessarily bad, as mentioned in [15], 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 1 is not a cause for alarm. We feel it should be a bit higher, but improving the practices should help fix that.

b. Final Tally

$$3.70+2.85+3.30+3.85+2.90+3.80 = 20.40 \qquad (2)$$

In the end, our agents got a score of 20.40 out of a possible 30 points, as seen in formula 2. Though we should note, since predictability is an attribute that we do not want to be maxed out, as explained in **??**, we did not include it.

$$\frac{20.40*100}{30} = 68 \qquad (3)$$

Using a rule of three to convert that to a more readable 100-point scale, as shown in formula 3 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.

VII. 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 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.

a. 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 a, 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

b. 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 **??** 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.

c. 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 each 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 b.

REFERENCES

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

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

[3] T. Kogod, "11 ways dungeons dragons influenced video games," TheGamer, Sep. 2020, acessed October 2022. [Online]. Available: https://www.thegamer.com/ways-dungeons-dragons-influenced-video-games/

[4] P. Team, "Prismarinejs," GitHub, acessed on October 2022. [Online]. Available: https://prismarine.js.org/

[5] M. Team, "Mineflayer," PrismarineJS, acessed October 2022. [Online]. Available: https://mineflayer.prismarine.js.org/#/

[6] M. pathfinder Team, "Mineflayer-pathfinder," GitHub, Oct. 2022, acessed October 2022. [Online]. Available: https://github.com/PrismarineJS/mineflayer-pathfinder#readme

[7] M. collectBlock Team, "mineflayer-collectblock," GitHub, May 2022, acessed October 2022. [Online]. Available: https://github.com/PrismarineJS/mineflayer-collectblock

[8] D. D. Team, "Overview," Docker Docks, acessed October 2022. [Online]. Available: https://docs.docker.com/get-started/

[9] ——, "Persist the db," Docker Docs, acessed October 2022. [Online]. Available: https://docs.docker.com/get-started/05_persisting_data/

[10] J. Carmack, "Readme.txt," GitHub, Dec. 1997, acessed October 2022. [Online]. Available: https://github.com/id-Software/DOOM

[11] E. Maiberg, "Why gamers are worried about 'minecraft: Windows 10 edition'," Vice, Jul. 2015, acessed Jan 2022. [Online]. Available: https://www.vice.com/en/article/53984z/why-gamers-are-worried-about-minecraft-windows-10-edition

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

[13] itzg, "Readme," GitHub, Oct. 2022, acessed October 2022. [Online]. Available: https://github.com/itzg/docker-minecraft-server

[14] D. D. Team, "Use volumes," Docker Docs, acessed October 2022. [Online]. Available: https://docs.docker.com/storage/volumes/

[15] P. G. A. P. C. M. A. Jhala, "Metrics for character believability in interactive narrative," Tech. Rep., 2013.