Institut für Informatik
Lehrstuhl für Programmierung und Softwaretechnik

LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor Thesis

MicroPsi and Minecraft

A Popular Video Game as a Simulation Environment
for a Cognitive Architecture

Jonas Kemper

Medieninformatik Bachelor

Aufgabensteller: Prof. Dr. Martin Wirsing
Betreuer: Joscha Bach PhD, Annabelle Klarl
Abgabetermin: 19. September 2013

Ich versichere hiermit eidesstattlich, dass ich die vorliegende Arbeit selbstständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

München, den 19. September 2013

. . . .

(Unterschrift des Kandidaten)

Zusammenfassung

Simulationsumgebungen spielen für das Testen und Erforschen von neuen Ansätzen für künstliche Intelligenz eine entscheidende Rolle. Da sich Computerprogramme nur mit erhöhtem technischen und finanziellen Aufwand in die physische Welt implementieren lassen, können simulierte Umgebungen schnellere, günstigere und reproduzierbarere Ergebnisse liefern. Genauso wie Innovationen aus dem Bereich der KI neue Impulse setzen, können auch neue Ansätze für Simulationsumgebungen zu neuen Erkenntnissen führen.

Das populäre Videospiel Minecraft bietet sich durch die kompositionale Semantik der Spielwelten besonders als Grundlage für eine Simulationsumgebung an, da das Agentensystem, durch das Erforschen seiner Umwelt, Wissen über die Spielwelt aufbauen und sich ähnelnde Strukturen wiedererkennen kann. Objekte der Spielwelt sind in Minecraft keine bloßen Hindernisse, sondern werden mit variablen Eigenschaften prozedural generiert. Durch die Komposition dieser Objekte, entstehen Umgebungen, die einer realen Umgebung in dieser Hinsicht eher gleichen, als andere virtuelle Welten. Da Minecraft einen umfangreichen Mehrspielermodus beinhaltet, lässt es sich zudem für Multiagenten-Umgebungen und so für Simulationen mit kollaborativen Agenten verwenden. Darüber hinaus sind Minecraft-Lizenzen günstig zu erwerben, erhältlich für viele Plattformen und es gibt eine äußerst große und aktive Community für selbsterstellte Spiel-Inhalte und Modifikationen.

Diese Arbeit präsentiert eine Schnittstelle, welche es der kognitiven Architektur MicroPsi 2 ermöglicht, sich an einem Minecraft Server anzumelden, die Umgebung wahrzunehmen und sich darin fortzubewegen. Im Anschluss daran, wird die Visualisierungskomponente vorgestellt, welche eine mit OpenGL gerenderte 3D-Ansicht der Simulationsumgebung live im MicroPsi 2 User Interface anzeigt. Es folgt die Beschreibung eines einfachen Experimentes, bei welchem sich ein Agent selbstständig auf ein vorher definiertes Objekt zubewegen muss. Die Arbeit endet mit der Dokumentation und Auswertung des Experimentes.

Abstract

Simulation environments play an important role for testing and researching new approaches to artificial intelligence. Because computer software can only be implemented into the physical world with increased technical and financial effort, simulated environments can deliver results faster, cheaper and more reproducible. In the same way as innovations in AI can deliver new impulses, new approaches to simulation environments can just as well lead to new insights into the future of intelligent machines.

Because of its compositional semantics, the popular video game Minecraft turns out to be an interesting simulation environment, as agents can generate knowledge through exploring their environment and recognising similar structures. Objects in Minecraft worlds are not just obstacles, but are generated procedurally with varying characteristics. Through composition of these objects, environments emerge that in this aspect are more similar to real environments than other virtual worlds. Since Minecraft contains an extensive multiplayer mode, it is also suitable for multi-agent environments and for simulations with collaborative agents. Furthermore, Minecraft licenses are affordable and available for many platforms and there is a huge and active community for player generated content and modifications.

This thesis presents an interface which enables the cognitive architecture MicroPsi 2 to connect to a Minecraft server, perceive its environment and move around within it. Subsequently, the visualisation component is introduced that displays an OpenGL rendered 3D view live in the Micro Psi 2 user interface. It is followed by the description of a simple experiment in which the agent has to move towards a previously specified object. The thesis ends with the documentation and evaluation of the experiment.

Acknowledgements

I would like to thank Prof. Dr. Martin Wirsing for giving me the opportunity to prove my motivation in writing a thesis about what I considered would suit me best. I thank my supervisor Joscha Bach, for letting me work with his project, advising and inspiring me through the entire process. I especially thank my supervisor Annabelle Klarl, for supporting my plan without hesitation, guiding me all the way and providing valuable feedback whenever I needed it. Dominik Welland helped me with understanding and building upon the MicroPsi framework and fixed inhabitant bugs as soon as I reported them, thank you!

Additionally I would like to thank Michael Fogleman whose code I built upon and especially Nick Gamberini who allowed me to use his framework `Spock` for this project and who answered every question about it in detail. Finally, I thank all the other members of the Minecraft community, who collect and distribute structured knowledge regarding the game in a combined effort and have answered my questions about the game and how it works on IRCs and message boards.

Contents

Chapter 1

Introduction

The hunt for artificial intelligence (AI) started many years ago. AI research is conducted for many purposes. A rough separation of these can be made, by separating the goals into developing useful, working applications and learning about the nature of intelligence itself. The ultimate task of the latter—recreating human intelligence—admittedly still seems to be science fiction.

A new generation of cognitive scientists, psychologists and computer scientists strives to implement new ideas for simulated minds by building cognitive architectures, which implement AI systems that are inspired by the human brain. Many of them do so by simulating, in one way or the other, what are called neural node nets. [GLA+10] To test the functionality of such cognitive architectures and to figure out their capabilities and potential, we need to study their behaviour inside defined environments. As implementing AI into the physical world (as robots, for example) requires patience and building appropriate hardware, computer-simulated environments play an important role.

One of these architectures is MicroPsi 2 [Bac12]. Developed by Joscha Bach and Dominik Welland, it is based on the Psi theory by Dietrich Dörner [Doe], a German professor for theoretical psychology. It aims at providing a reliable and complete implementation of his theory of cognition while at the same time being easily accessible, understandable and modifiable for the research and applications to come. MicroPsi 2 is both: a cognitive architecture as well as an interface to a set of different simulation environments.

1.1 Motivation

Video games are natural applications of artificial intelligence. The quality of a game's AI can make the difference between an immersive experience and an uninspired demonstration of computer graphics technology. One game that especially stood out in the recent years is Minecraft. As a so called *sandbox game* [Dun11], it attempts to leave the player with as few limitations as possible for what they do and how they behave in the game world [Squ07]. This kind of open world delivers a simulated environment rich of opportunities, for both human and artificial players.

Even though MicroPsi 2 already has integrated simulation environments, it is aimed for to experiment with offbeat ideas for AI simulations to generate new insights and opportunities for future research. Building an interface in between the cognitive architecture MicroPsi 2 and the video game Minecraft is what this thesis is about.

1.2 Outline

This project is fundamentally about combining existing technologies. The most important ones being MicroPsi 2, the framework aiming to implement the ideas of the Psi theory, and Minecraft, the popular sandbox videogame.

To understand how and why they were chosen, a brief history of their evolution as well as explanations of their basic ideas and relevant insights into their architecture are given in chapter 2 and 3. Chapter two introduces Artificial General Intelligence and in particular the Psi theory and its implementations. We will focus on the particular implementation MicroPsi 2 and describe its modules in detail. Chapter 3 introduces the video game Minecraft, its protocol and related software projects. As the resulting software heavily relies on them, the protocol and the utilised open source software will get increased attention. In Chapter 4 the contribution of this thesis, the Minecraft interface for MicroPsi 2, is being described in detail. As a proof of concept, the description eventually leads to a small experiment. Chapter 5 summarises the findings of this thesis and gives an outlook on possible future applications.

Chapter 2

Artificial General Intelligence

To provide the necessary context for this thesis, this chapter will provide a brief introduction to the corresponding foundations on the AI side. First, we will recap the relevant terms and definitions for this project. Next, we will introduce Artificial General Intelligence (AGI) as a subset of AI. Based on this, the Psi theory, as a particular instance of AGI theories, is described in detail. The description includes a brief history of its concrete implementations. Then, MicroPsi 2, the latest framework dedicated to the Psi theory, is described in detail.

2.1 Agents and Environments

Before we continue to work with concrete theories and implementations, we define agents and environments according to Russel's and Norvig's standard reference *Artificial Intelligence: A Modern Approach* [Rus09]. An *agent* is defined as anything that can perceive its environment through sensors and conduct actions inside the environment through actuators. The examples given in [Rus09] include a software agent that receives network packets as sensory inputs and produces output by writing files and sending responses. However, agents can just as well be robots that have cameras as sensors and motors as actuators. Even humans can be thought of as agents that perceive their environment through their senses and act upon it through their muscles.

Russel and Norvig furthermore define the *percept*, as the agent's input and the *percept sequence* as the complete history of what the agent has perceived. Each decision an agent makes is based on the percept sequence. Additionally, they describe the *agent function*, as the function that maps each percept sequence to an action. The *agent program* is its concrete implementation.

Environments have a number of different properties. They can either be *fully observable* or *partially observable*. An environment is fully observable when the agent has access to the complete state of the environment—or at least to everything that is relevant, regarding to its performance measure. As opposed to this, it is partially observable, if the previously mentioned requirement is not given.

An environment can moreover be *single agent* or *multiagent*. An environment is multiagent, when there is at least one other agent, whose behaviour is about maximising a performance measure that depends on the first agent's behaviour. If the agents try to maximise the performance measure of all agents, the environment is called *cooperative*. If agents try to maximise their own and minimise the performance measure of the other agents, it is called *competitive*.

A *task environment* is the combination of the performance measure, the environment and the agent's actuators and sensors.

2.2 Strong AI

Looking upon the field of AI from a philosophical point of view, computers seem to deliver enormous potential for learning about how minds work. At the same time, new questions arise. If we knew how cognition works and if we could build machines that simulate it, would these minds be real? And what would the ethical implications be? According to Russel and Norvig [Rus09], one can distinguish in between two fundamental assumptions. The assumption that computers are able to act *as if* they were intelligent is called the weak AI hypothesis. Thinking that an intelligently acting machine, in fact, *is* performing cognition, is called strong AI hypothesis. Putting it differently, the weak AI hypothesis considers computers to be an instrument to research cognitive processes, whereas the strong AI hypothesis considers simulated cognitive processes as actually being cognition.

It took many years for AI research to evolve from the early ideas of thinking machines over Deep Blue—the computer that could beat mankind's best chess players—to Watson, the AI that beats the champions of Jeopardy, a game show about asking the appropriate question to a given answer. There exist many applications for AI: self-driving cars and online-shopping recommendation systems to name a few. These examples have one thing in common. They are applications of technology that serves an immediate, or at least foreseeable purpose. An actual (hypothetical) implementation of a strong AI would instead mean one would have to build a machine that is capable of acting like a human being—not just for a limited problem set, but in all of them.

Another term that is being used more recently as a synonym for strong AI is *AGI*, short for *Artificial General Intelligence*. It was coined to delimit itself from AI applications such as chess computers and other expert systems. AGI sets out to develop software that can solve and act appropriately in a wide variety of problem fields, without specialising on any particular problem. A complete AGI system is supposed to control itself autonomously and have its own thoughts and even feelings—depending on the definition of these. This has been the original focus of the AI field, before many researchers lost their enthusiasm when it turned out to be not as imminent as expected. Even though the advances in other AI disciplines contribute to it, AGI is more than just an assembly of existing applications. There is a huge number of different AGI projects being worked on, with most of them being in early stages. Popular examples include the OpenCog[1] project that is developed as a part of the Google Summer of Code projects and Grok[2], which is the commercial product and venture of Palm founder Jeff Hawkins. [GP07]

Cognitive AI, as a particular approach to AGI, can be understood as architectures that are inspired by the interdisciplinary field cognitive science, psychology, and the neurosciences and implement recent theories and findings from these disciplines. The purpose of these architectures is to see if the theories hold against what they promise. Many cognitive architectures share characteristics with or directly implement artificial neural networks.

The most famous indicator for whether a machine is intelligent or not, is the Turing test. Decades after its formulation, contests regarding to it are still being held—may

[1] `http://opencog.org/`

[2] `https://www.groksolutions.com/`

the best conversationalist win! Many would argue that even an algorithm that passes the Turing test would still not be intelligent. It would trick people into thinking it was, but it would still not be conscious, aware of itself. Moreover, it would not have emotions. [Rus09]

2.3 Psi Theory

This section gives an overview of the basic ideas of the Psi theory. The descriptions are based on *Principles of Synthetic Intelligence PSI: An Architecture of Motivated Cognition* [Bac09] by Joscha Bach. The theory in its foundations was first described by German psychologist Dietrich Dörner in his books "Bauplan für eine Seele" [Doe] and "Die Mechanik des Seelenwagens" [DaS02] from 1998 and 2002.

As a psychologist Dörner saw a problem with the common methods of his discipline. In his opinion, they are not suitable for finding explanations to the most burning questions about how human behaviour and awareness are formed. He tries to deliver a holistic approach that goes beyond classic problem solving and develops a unified model for cognition that implements motivation and emotions as central components. He is convinced that artificial intelligence does not have to focus on different aspects of cognition that have to be looked at separately but that a unified theory will ultimately lead to a deeper understanding of the mind itself.

Dörner's "Bauplan für eine Seele" [Doe] ("blueprint for a mind") belongs in the interdisciplinary field cognitive science, as with his theory he tries to research psychological foundations with the methods of computer science. The theory is an attempt to explain the mind as a machine. It is looked at as a fuzzy and self-extending causal network structure.

Psi agents are controlled by a structure of relationships and dependencies, that strives to maintain homeostatic balance. Every form of representation in the agent's cognition—may it be percepts, plans or abstractions of space and objects—is represented as directed, hierarchic spreading-activation networks. Thes structures can be visualised an artificial neural networks, which we will call *node nets* in the following. Nodes of these node nets have gates that may send activation and slots that may receive it. The basic elements a node net consists of are called *quads*. Quads themselves consist of one central neuron and four outer neurons that are called *sub*, *sur*, *ret* and *por* (see figure 2.1). The outer neurons can be connected with the central neurons of other quads and thereby form networks. The four different kinds of outer neurons represent different kinds of relations. *Sub* stands for a "has part" relationship, *sur* is the inverse to sub an represents an "is part" relationship, *ret* stands for the ordering relationship of succession and *por* represents predecession.

Furthermore each neuron has an activation value, that it receives from and forwards to other neurons. Links in between neurons can be modified to dynamically modulate the forwarded activation value. The modulation of the flow of activation is interpreted as emotions. There are specific kinds of neurons that are called *sensors* and *actuators*. They represent the in- and output to the outside world. Sensors receive activation if they are triggered by the environment and activated actuators lead to an according action in the environment.

The motivation of Psi agents comes from a set of predefined drives. They are divided into physiological (e.g. physical integrity), social (e.g. affiliation) and cognitive drives (e.g. reduction of uncertainty). These drives may signal a specific demand. If the value of a demand changes, pleasure or displeasure signals are sent to the agent. By

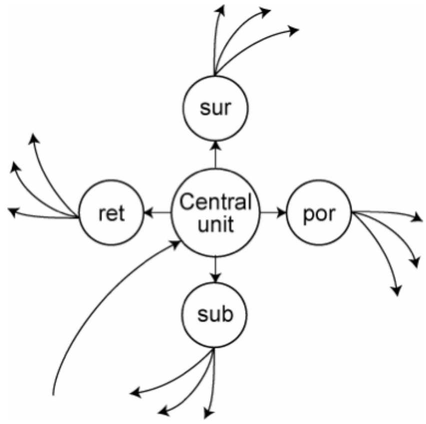

Figure 2.1: *Quads like this are the basic representational unit of the Psi theory (taken from [Bac09])*

reinforcement learning agents learn what behaviour leads to the fulfilment of demands.

As node nets may create new nodes and links on their own, the agent forms memory, motives and plans, that help it to select what action to execute next (see figure 2.2). Agents can store representations of their entire percept sequence, but only those that are connected to pleasure or displeasure signals are being kept.

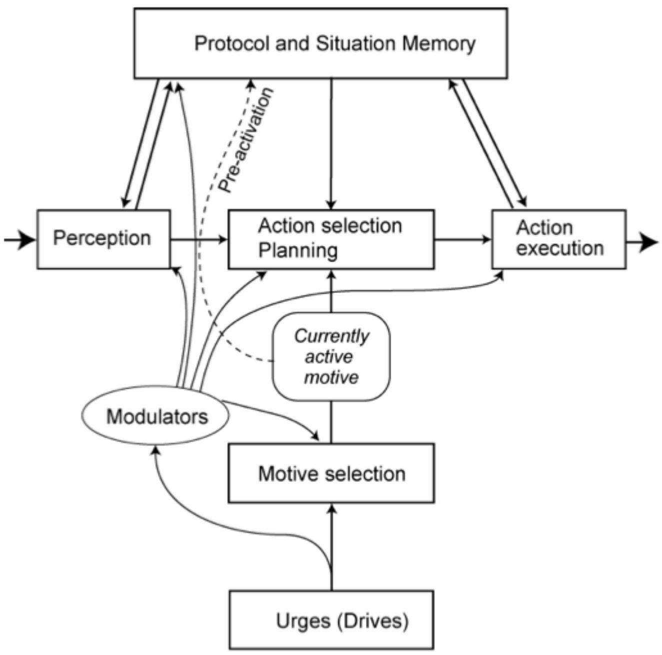

Figure 2.2: *The basic architecture of action selection in Dörner's theory (taken from [Bac09])*

Quads are used to form *partonomies* that represent information about concepts and relationships. A partonomy is a hierarchical structure that represents a part-whole relationship. For example, the concept "head" would be represented as the assembly of a face and a head. A face furthermore would itself consist of a mouth, a nose, eyes and so forth (see figure 2.3).

Through the combination of partonomies and order relations, all basic types of structured information the agent stores are formed. These could be causal, spatial or temporal relationships in between states of the environment, for example.

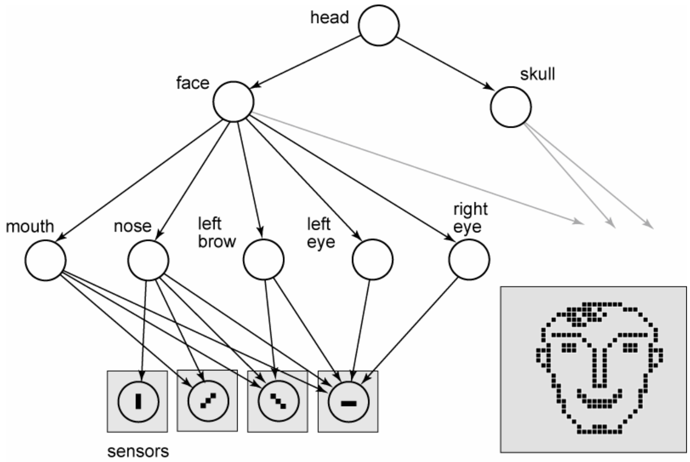

Figure 2.3: *The partonomic representation of a Cartoon face (taken from [Bac09])*

Psi agents' basic approach to problem solving works by going through a set of subsequent stages. The agent first tries to approach a problem with behavioural routines. If these do not solve the problem, the agent tries to construct a plan, by trying to find a chain of causal relationships that ultimately lead to a state, that solves the problem. If no plan is found, the agent starts to explore its environment to gather more knowledge.

The goal of implementing the theory to computer software is to find potential inconsistencies and moreover to find out if Psi agents show similar behaviour to that of humans. If they do, the Psi theory could possibly serve as an explanation about how human minds work. For this purpose, a number of experiments—similar to video games—have been concluded, in which both Psi agents and human players had to perform specific tasks. Later on, their behaviours have been compared and searched for similarities and differences.

Joscha Bach adapted the theory to bring it in a contemporary form with slight modifications in his dissertation *Principles of Synthetic Intelligence PSI: An Architecture of Motivated Cognition* [Bac09]. Even though building a conscious machine that thinks and acts as we do is still mere science-fiction, it is this kind of foundational research, that leads us to new ways of thinking about the world, that give us our most significant leaps.

2.4 Psi Implementations

Psi has been implemented by different groups at different times. The first implementations are by Dörner and his associates themselves (see figure 2.4). They used Delphi Pascal and developed it for Windows environments. This implementation can still be downloaded[3] and runs on Windows 7 installations, for example.

The work on Dörner's team's software has not been continued, so Joscha Bach and his associates developed new implementations of Psi. From 2003 to 2009 they built an implementation in Java as a set of plugins for the Eclipse IDE called MicroPsi. It

[3] `http://www.uni-bamberg.de/psychologie/theoretische-psychologie/leistungen/forschung/downloads/`

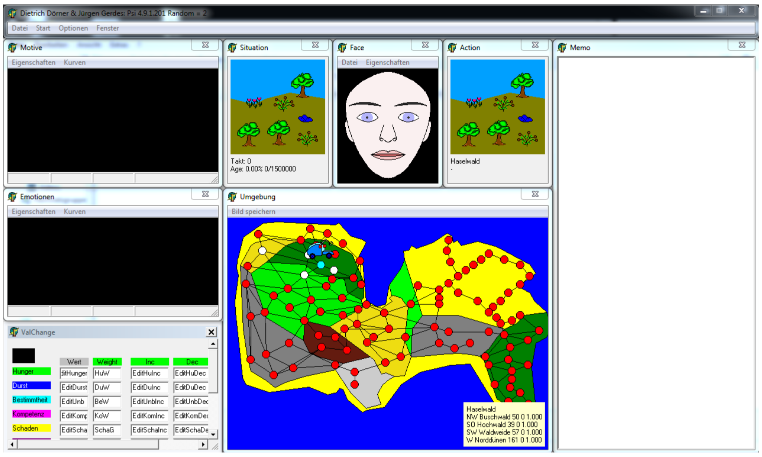

Figure 2.4: *Screenshot of Dörner's team's Delphi Pascal Psi implementation*

included a graphical editor and a broad experimentation framework. MicroPsi provided a complex and sophisticated simulation component which contained simulated objects in three dimensional worlds—even though most experiments took place on a flat surface. The framework offered complex administrator interfaces and appealing DirectX rendered 3D-views of the scenery as well as a more general world view which provided a graphical user interface to the world by providing clickable objects. Predefined simulation environments included a classic Island and a Mars world (see figure 2.5). Objects in these worlds may be assembled recursively out of other objects and bring their own functionalities. As it has been the case in Dörner's implementation, a viewer for facial expressions has been included—this time as a three-dimensional simulation, too. [Bac09]

Figure 2.5: *Screenshots of the 3D visualisation component in MicroPsi (taken from [Bac09])*

Simulated environments proved especially to serve well for the research of collaborative behaviour of multiple agents, for mapping and exploration, image processing as well as for memory and planning. Additionally, some scenarios, that are almost impossible to implement in the physical world (such as evolving agent populations) could easily be simulated. On the other hand, downsides of simulated worlds include being limited by computing power and the programmer's specifications. [Bac09]

2.5 MicroPsi 2

To ensure broad understandability and to maintain platform independence, MicroPsi has been built ground up again in 2011 and 2012 using more lightweight Python code. What is remarkable about the new implementation called MicroPsi 2 (in the following MicroPsi), is that the simulation is deployed as a web application and the graphical interface is completely rendered inside a web browser, using state-of-the-art internet- and web application technologies. [Bac12]

Even though there have been more complex simulation environments (e.g. 3D-worlds) for previous implementations of Psi architectures, the relatively new version of MicroPsi so far has only two fairly simple ones: a 2D-Island and a map of the public transportation system of Berlin (see figure 2.6). Instead of building a new 3D-world, with this project we set out for something more experimental. More on this follows in chapters 3 and 4.

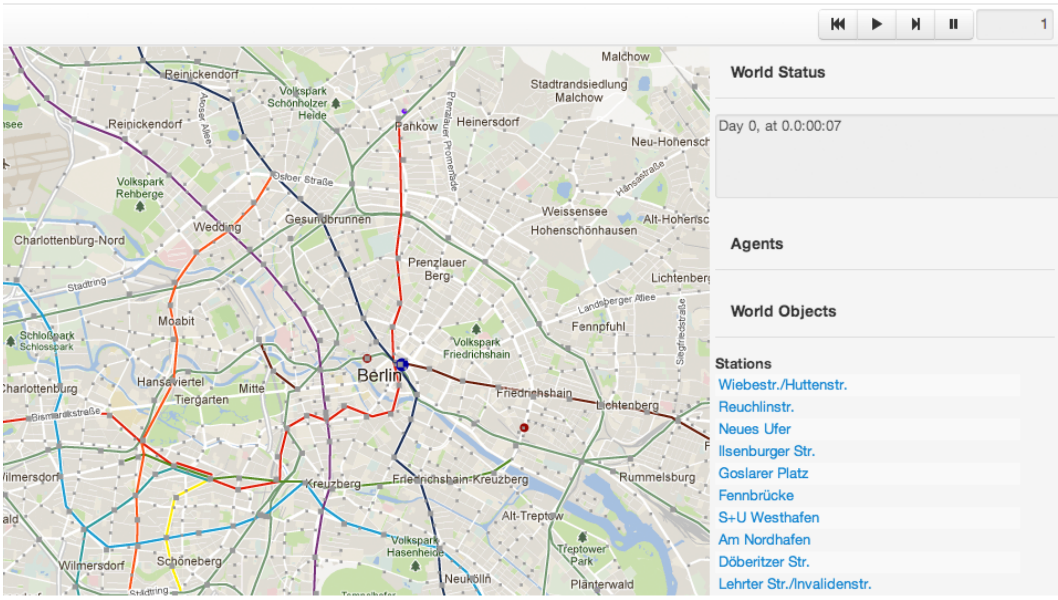

Figure 2.6: *MicroPsi's simulation environment Berlin*

2.5.1 Module Overview

MicroPsi is written in Python with a minimum of dependencies. Therefore, its modular structure is comparably easy to understand. It is illustrated in figure 2.7. First, one can differentiate in between the `Server` module (or the web interface) and the actual simulation `Runtime` module (also called *core*). In a simple simulation experiment setup, MicroPsi runs three threads: one for the `Server` and, invoked by the `Runtime`, one *world runner* that runs a simulation world as well as one *node net runner* that runs a node net. As these names suggest, the `Runtime` manages both the simulation environments as well as the inhabitant agents (or node net embodiments). They may by design run asynchronously. In fact, the `Runtime` works entirely independent of the `Server` and therefore may just as well be deployed for command line interaction or other GUIs. Furthermore, the `Server` contains a `UserManager` and the `Runtime` a `ConfigurationManager` that provide functionalities for multiple users to create accounts and individually create, run and store simulation setups. [Bac12]

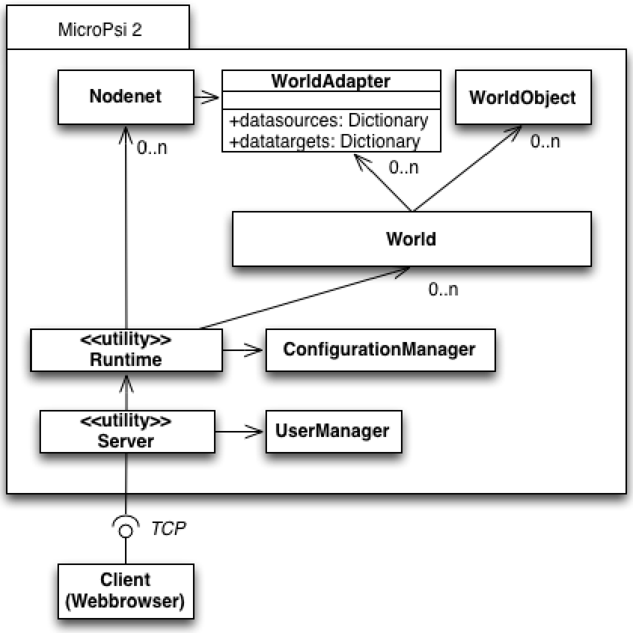

Figure 2.7: *The modular architecture of the MicroPsi framework makes it easy to extend. (taken from [Bac12])*

The following description is heavily based on [Bac12], where the theoretical foundations can be found in detail.

2.5.2 Module `Server`

The `Server` renders the GUI and deploys the agent simulation as a web application. It acts as a web server for remote or local access. A client for this application may be any computer with a reasonable up-to-date web browser. Therefore, simulations can be launched from anywhere without requiring any installation. It rests upon the lightweight Python web framework *Bottle*[4].

The web interface is based on HTML as well as Javascript. The communication in between the browser and the simulation is managed via JSON remote procedure calls. Many GUI components of Twitter's *Bootstrap*[5] library are in use. The graphic renderings (see figure 2.8) utilise the JavaScript graphics library *PaperJS*[6].

The `Server` communicates with its users through the server API. User sessions and access rights are managed by the `UserManager`.

2.5.3 Module `Runtime`

In this setup, the `Server` starts the `Runtime`—even though it may also work independently of the `Server`. The `Runtime` component communicates with the `Server` through the MicroPsi API. It manages node nets and worlds.

[4]`http://bottlepy.org/docs/dev/`
[5]`http://getbootstrap.com/`
[6]`http://paperjs.org/`

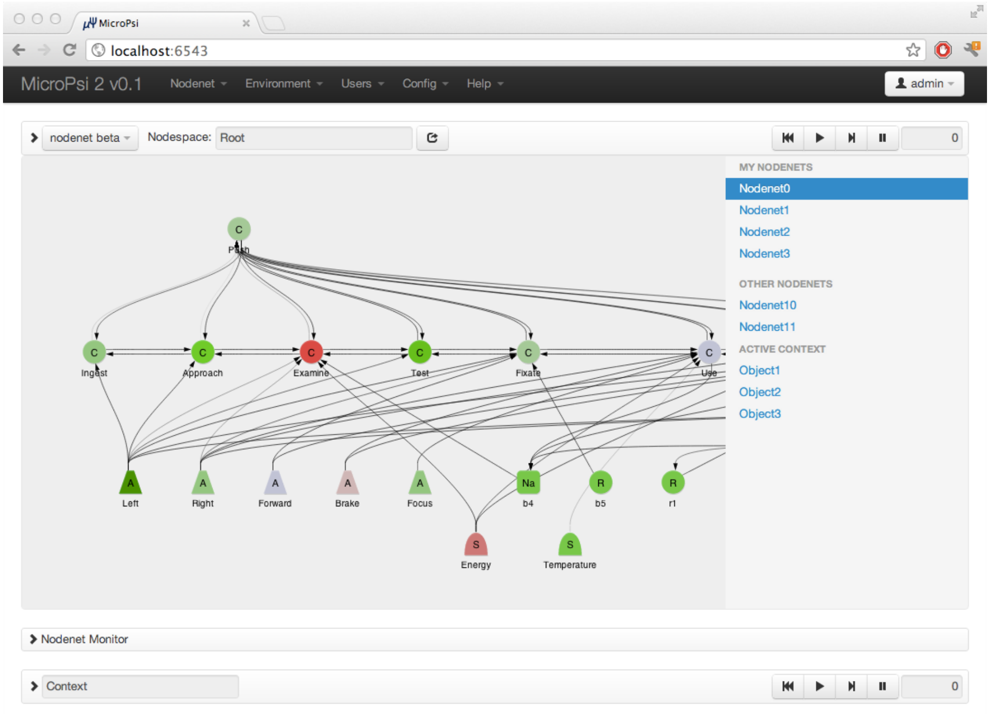

Figure 2.8: *The graphical editor is the primary interface to node nets. (taken from [Bac12])*

2.5.3.1 Node Nets

MicroPsi defines agents as node nets or to be more specific, hierarchical spreading activation networks. They are an "abstraction of the information processing provided by brain" [Bac12]. Agents can be placed and researched in simulation environments or physically embodied as robots.

A MicroPsi node net is defined as a set of states, a starting state, a network function, that defines how to advance to the next state, and a set of node types. Data sources and data targets serve as the input for nodes and output towards a world. As such, data sources are filled with data from the world and data targets are linked to agent actions in the world.

According to the Psi theory, nodes may have different types and parameters. They contain gates and slots that send and receive activation. In most cases, the activation is forwarded from a slot to a gate without further modulation. However, nodes may include functions that enable the creation of new nodes and links as well as procedures for learning and planning. They may be implemented as Python code.

2.5.3.2 Worlds

The simulations worlds are the environments in which we can study our agent's behaviour. Worlds need to provide a world adapter which functions as the interface in between a node net and the environment. Within the world adapter, data sources and data targets have to be defined carefully, to get a functional and meaningful experiment going. They represent the agent's sensory input and the motoric output. Sophisticated interconnection of those enables interaction with the environment.

The *world adapter* has to define *data sources* and *targets*. It fills the sources with data from the world and writes the targets to the world. The node net does the opposite: it reads from the sources and writes into the targets. This enables a feedback loop in between the world and the node net. Furthermore, the *world adapter* provides a step function, that advances the world and is called by the MicroPsi world runner frequently.

The kind of data the world adapter interfaces, is not specified any further, which gives developers the opportunity to experiment with classic simulation worlds, as well as exotic applications (eg. stock data). At the time of the original development of the MicroPsi, the prioritised application was building a framework for knowledge representation.

Objects
Worlds contain objects. Objects may be anything that could be interesting for a simulation, and that needs some kind of integrated logic. Light sources or collectable resources are a common example. Objects may contain a set of functions and states, but at least one function that determines how the object advances and reacts to changes while moving through a simulation cycle. A comprehensive world function calls every object function for each simulation step of the world.

Agents
Agents are objects that are connected to a world adapter which makes them controllable by a node net. The object that incarnates the agent is best thought of as the agent's body. The function that advances the agent checks for input from the node net and sends its output to it.

2.6 Summary

This chapter provided the necessary foundations regarding AI for this Thesis. It introduced Artificial General Intelligence as a subset of AI research and presented its main ideas and goals. Then, the Psi theory by Dietrich Dörner has been described briefly. An overview of implementations of the Psi theory has been given. The latest framework dedicated to the Psi theory, MicroPsi 2, has been described and its architecture outlined. Knowing about MicroPsi's modules is a necessary foundation for the additions to it that are described in chapter 4.

Chapter 3

Minecraft

To understand how and why the video game Minecraft[1] is expected to be an interesting simulation environment for artificial general intelligence, necessary background information, about what makes Minecraft different from many other games, has to be given first. To furthermore work with it, the essential components of the game's architecture have to be described. This chapter provides a roundup of Minecraft's history and a brief explanation of the basic game mechanics. They are followed by an overview of Minecraft usages that exceed the original game's purpose as well as an explanation, of why Minecraft is suitable as a simulation environment. Eventually, related software projects, that have been used for this thesis are introduced and described.

3.1 Overview

The story of Minecraft has many interesting sides, but first and foremost it is a story of immense, unexpected success. To give an overview of it, this section delivers insights into its central ideas, as well as a general look upon the developer's practices. Moreover, the game's reception is considered, as the game has become unusually popular. The descriptions are based on Sean C. Duncan's article *Minecraft, beyond construction and survival* [Dun11].

3.1.1 Main Ideas

When Markus ("Notch") Persson built and released the first public version of Minecraft, it soon became apparent that his creation resonated with many people. The simple concept of a world entirely consisting of standard sized building blocks, which the player can create, destroy and relocate one-by-one, enabled many gamers to employ their creativity, explore the Minecraft world and test out its possibilities and boundaries.

Looking at the game for the first time, it is impossible to oversee the primitive appearance of its graphics engine. Everything in the game is made out of blocks. May it be the leaves of a tree, dirt, people or the clouds. Even the sun does not appear round, but as a cubic object. One can assume that the obvious connotation to Lego™ is more than a coincidence, as it immediately gives the player a clue, about what kind of possibilities for building and creating the game might deliver.

Another interesting aspect about Minecraft is the procedural semantics the game world is generated with. Trees in Minecraft, for example, may share a similar structure that consists of a trunk and leave-covered branches spreading out fractally, but the

[1] `https://minecraft.net/`

particular characteristics of each tree are generated randomly. This makes a Minecraft world somewhat more realistic than those of many other videogames.

Minecraft strives to be the opposite of a game with a strict game-flow. Instead, it enables the player's creativity and joy of creation.

3.1.2 Development Process

The development process of Minecraft is very different from those of most other well-known games. Instead of building the game over months or years to finally release a (more or less) finished version to the public, Minecraft is being designed iteratively. In fact, Persson released the first version only a week after he started working on the game. Ever since, new features have been added, and new versions of the game have been released frequently. Another defining aspect of the approach of the game's developement studio Mojang approach towards the development of Minecraft is the involvement of the game's players. Since the earliest stages, plans for the game's future have been discussed openly, and many times the players' voices have been heard. Persson himself locates his approach in the field of agile software development. [Dun11]

3.1.3 Reception

The game has attracted great attention ever since its first release in 2009. Since copies of the game could be obtained commercially for the first time, its different versions sold more than 26 million times—with the PC version priced at 19.95€, for example. It should be noted that Minecraft's development studio Mojang is a so called *indie game developer*, that is not associated with any classical game publisher, but distributes copies of their game exclusively through their own website.

Minecraft can certainly be called a world wide phenomenon. The game's developers received a number of prizes for their creation—not to mention the millions of dollars that have been earned in Minecraft sales. With its huge success, Minecraft has proven clearly that it compelled to many players.

Unlike any other game, Minecraft fosters the creativity of its players. Bringing the construction component to one extreme after the other, well publicised player creations include a life-size model of the USS Enterprise-D as well as a fully functional arithmetic unit. Videos of roller coaster rides through complex constructions increasingly show characteristics of an art form themselves—regarding to the amount of views on youtube, not less than other, well established formats.

3.2 Game Mechanics

Now knowing about the game's background, in this section we will continue with descriptions of the basic concepts of the game, that regard to our AI interface.

Before a new Minecraft game begins, it procedurally generates a three-dimensional world. The player is then dropped at a spawn point and is now facing a scenario and a minimal game interface, lacking any kind of instruction or advise about what to do next. Soon, the procedurally generated world drives most players to explore the highest mountains and deepest caverns. As the player collects resources and crafts items, resources harder to reach and items more complex to craft become available. [Dun11]

Although the game can be downloaded and played as a single packet of software, many scenarios of playing the game consist of running a Minecraft server software, as well as one copy of the client software for each player. It is possible to mimic the official client by implementing the reverse-engineered client server protocol and therefore build artificial players that way.

The structure of Minecraft worlds and the client server protocol are described in the following. The descriptions are based on the regarding articles of the *Minecraft Wiki*[2].

3.2.1 Minecraft Worlds

Figure 3.1: *A grass block (CC-BY-3.0 Mojang AB) [ima13]*

The most significant concept in Minecraft is the block (see figure 3.1). A block is a cube which sides are, compared to the player, roughly one meter long. Every Minecraft world is built up entirely out of them. Therefore, worlds can be thought of as three dimensional spaces filled with voxels. [BB] There are many different types of blocks (or materials they consist of) and they share their single size, which converts to the basic distance unit of Minecraft.

A chunk (see figure 3.2) is a segment of the Minecraft world that is 16 blocks long, 16 blocks wide and 256 blocks high (or deep) and therefore consists of up to 65,536 blocks. [mcw13]

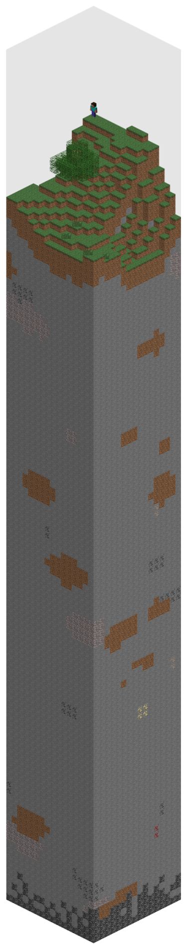

Figure 3.2: *A chunk (CC-BY-3.0 Mojang AB) [ima13]*

"The player" (see figure 3.3) is what the playable game-character in Minecraft is called. It is usually displayed as a human. Also, a Minecraft world has a day-night-cycle with 24 Minecraft hours converting to 14 minutes by default.

The game itself has no predefined goals. Players can walk around, discover the created world (see figure 3.4 ①) and collect resources by *destroying* blocks, with the generated resources equaling the block type. They can combine several resources to *craft* items. For example, a player can destroy the blocks that represent a tree (see figure 3.4 ②). The gained *wood* resource could then be used to craft a wooden pickaxe (see figure 3.4 ③), which could then be used to dig into the ground more efficiently (see figure 3.4 ④) to then *mine* more rare resources, like iron or gold.

There exist different game modes. The original *survival* mode adds monsters that attack the player at night. What solutions to survive the player comes up with (e.g. building shelter or fighting the monsters) is left up to him or her. In *creative* mode, the player is not being attacked by monsters, has the ability to fly and is given instant access to unlimited resources. The mode of a Minecraft world does not effect the functionality of this project.

3.2.2 The Client Server Protocol

Minecraft's client server protocol is not publicly documented by the developers themselves. However, the modding community gathered comprehensive knowledge and understanding of its structure (probably by using reverse engineering techniques). The protocol is based on packets. Packets are either *server to client*, *client to server* or *two-way* and begin

[2] www.minecraftwiki.net

Figure 3.4: *Screenshots of Minecraft's basic game mechanics*

with a *packet ID* byte. The structure of the packet's payload depends on its packet ID.

To give an example of one of the easier packets, the *Client Position* packet is fairly straightforward (see table 3.1). It is exclusively send from clients to servers and starts with its packet ID (as every packet does), followed by the X- and Y-coordinates as doubles, the stance value as a double (which is used to modify the player's bounding box), another double for the Z-coordinate and eventually a boolean that describes if the player is on the ground or not. [pro13]

Field Name	Field Type	Notes
Packet ID	Byte	0x0B
X	double	Absolute position
Y	double	Absolute position
Stance	double	Used to modify the players bounding box
Z	double	Absolute position
On Ground	boolean	Derived from packet 0x0A

Table 3.1: *The structure of the packet "Player Position (0x0B)" [pro13]*

Figure 3.3: *The player (CC-BY-3.0 Mojang AB) [ima13]*

Knowledge of this data structure is already sufficient to move around in the Minecraft world. To go forward, one has to figure out the players current position, calculate the absolute coordinates of the destination of the movement in regard to it and send a Client Position packet with these coordinates via a TCP socket to the server. If the destination is not more than 100 blocks away from the origin of the movement, the server accepts the packet. In the official Minecraft client, a player's movement from one point to the other is rendered with a walking animation.

Other than movement, packet structures are defined for every aspect of the game. May it be the initial handshake, the creation or destruction of blocks or activities of other player- or non-player-characters.

This protocol is what a custom Minecraft client needs to speak.

3.3 Minecraft as a Gaming Platform

With more and more applications that move beyond the original gameplay of construction and survival, Minecraft seems to shift gradually towards being not only a game, but also a gaming platform, that many other people use to implement their own ideas. Inside Minecraft worlds, players have created games for educational purposes, for example. A group of students of the Miami University developed a game called Circuit Magic that is committed to teaching the players about logical circuits. The famous Minecraft Teacher, who implements Minecraft in primary school classes, is another good example. Following in his footsteps, "Massively Minecraft" is a community of teachers who share their thoughts and experiences with doing the same. Just as well, games have been developed for the purpose of artistic expression. Chain World, the game that tried to simulate religion, is an exciting example. Not only do players implement their engagement inside Minecraft. The internet is full of Minecraft content, be it videos on Youtube, online tutorials or other community based knowledge resources, headquartered at the Minecraft Wiki[3]. [Dun11]

As user-generated content has been an essential part in the development of Minecraft, in a process that reminds one of a self-fulfilling prophecy, more and more people explore its possibilities to experiment with new applications. As of today, Persson and Mojang seem to favour and approve new uses of their game. [Dun11]

3.4 Suitability of Minecraft as a Simulation Environment

There are a number of reasons why using Minecraft as a simulation environment could be useful and lead to interesting results. First, the game itself is easily accessible. It is developed using Java (for both the client and the server software) and, therefore, up to a certain extend, platform independent. The desktop computer version is being sold for Windows, Mac OS X and Linux devices. There are official ports for Android, iOS, Xbox 360, the Raspberry Pi and a version for the upcoming gaming console Xbox One is announced. The desktop versions are priced at 19.95 €, which make it affordable to a large audience.

The game itself already has an enormous fanbase. It is (like most videogames) especially popular among teenagers. Minecraft being loved by so many people could benefit this project, in terms of leading to increased attention.

The game's developer has proven many times that it acts generously towards other developers, when it comes to the creation of game modifications and content that builds upon or changes original Minecraft intellectual property. As mentioned before, Mojang is not restrictive towards users doing all kinds of things with their creations. This led to the availability of a fairly complete community-sourced documentation and explanation of virtually every aspect of the game—including its software architecture,

[3] `http://www.minecraftwiki.net/`

data structures and protocols. This is useful for this project, as chances are low that they will have anything against using Minecraft for AI simulations in the foreseeable future. To the contrary, the game's AI creator Jon Kagström provided advise for this project

The Minecraft world, with its logic, semantics and functionalities, offers possibilities for an AI to prove being able to interact with the environment—in primitive ways (e.g. moving around), as well as with increasingly complex tasks like building, collecting resources, crafting items and interacting appropriately with both well-disposed and hostile other entities.

The semantics of the gameworld share characteristics with the real world. Moving through a Minecraft environment, one quickly realises that the game has generated different biomes (eg. forest or tundra). Also, trees, rivers, mountains and ore veins are neither hard-coded nor appear completely random, but are generated procedurally and their structure appears to be (somewhat) fractal.

Using Minecraft as a simulation environment will give Psi agents possibilities to show off, what kind of sophisticated behaviour they are capable of.

3.5 Related Projects

Minecraft's success inspired many other projects, including artificial players and a number of Minecraft-like games in a wide variety of programming languages and environments.

3.5.1 Minecraft Bots

There exist many projects that could be considered Minecraft *bots*. One has to differentiate in between two types. On the one hand, there are those that mimic an entire client software and facilitate communication with the server on the default client software's behalf. On the other hand, there are bots which are modifications of the original client (or server) software and usually add non player characters—like animals and other non-human creatures—to the game. The code is usually injected through one of the popular *modloaders* (e.g. Minecraft Forge).

One example (and probably the most advanced one) for an entire bot framework that replaces the client, is *Mineflayer*[4]. It has a high-level abstraction of the environment (eg. entity knowledge and tracking) and is written in JavaScript using node.js. However, it has not been used for this project, because a Python implementation was aimed for, as it is explained in the next section. Opposed to Mineflayer, an example for game modification bots are the *Cubebots*[5]—fan-made non-player characters that aim to help Minecraft players with mundane tasks. Most third party clients, including bots, facilitate their own event loops.

3.5.1.1 `Spock` by Nick Gamberini

Developed by Nick Gamberini, `Spock`[6] is an open-source bot framework (and as such also a Minecraft client) written in Python. It has been chosen to become an essential part of this project for two reasons: being written in Python it painlessly integrates in

[4] `https://github.com/superjoe30/mineflayer`
[5] `http://www.minecraftforum.net/topic/1675965-152sspsmp-cubebots`
[6] `https://github.com/nickelpro/spock`

the existing MicroPsi code and the absence of dependencies (with one exception) leave the code understandable and easy to deploy.

3.5.1.2 Protocol Implementation in `Spock`

The Minecraft protocol implementation in `Spock` is straightforward. The necessary data structures are stored separately and can be accessed globally (see figure 3.5).

```
names = {
    0x00: "Keep Alive",
    0x01: "Login Request",
    0x02: "Handshake",
    0x03: "Chat Message",
    ...

structs = {
    #Keep-alive
    0x00: ("int", "value"),
    #Login request
    0x01: (
            ("int", "entity_id"),
            ("string", "level_type"),
            ("byte", "game_mode"),
            ("byte", "dimension"),
            ("byte", "difficulty"),
            ("byte", "not_used"),
            ("ubyte", "max_players")),
    ...
```

Figure 3.5: *An excerpt of `Spock`'s data structures for packet-IDs and -structures*

These structures are used to parse each packet appropriately (see figure 3.6). All data that is received from the server through the established TCP connection is first checked for a packet ID. Then, according to the ID the decoder can look up the appropriate data structure and may then parse the remaining fields of the packet until it reaches its end and the next packet ID is to be expected to arrive through the socket.

```
def decode(self, bbuff):
    #Ident
    self.ident = datautils.unpack(bbuff, te')

    #Payload
    for dtype, name in ta.structs[self.ident][self.direction]:
        self.data[name] = utils.unpack(bbuff, dtype)
```

Figure 3.6: *`Spock`'s function for decoding packets*

3.5.2 Minecraft Clones

Other interesting projects include Skycraft [sky13], a Minecraft-like browser game based on WebGL. Another particular project that has the same name as the original game that inspired it is `Minecraft` by Michael Fogleman (see figure 3.7). It is a simple Minecraft clone in under 600 lines of Python and has gained some popularity on reddit[7] and Hacker News[8]. It is comparably easy to understand and modify and has been used for the visualisation component of this project. It is based on the Python multimedia library *Pyglet*[9].

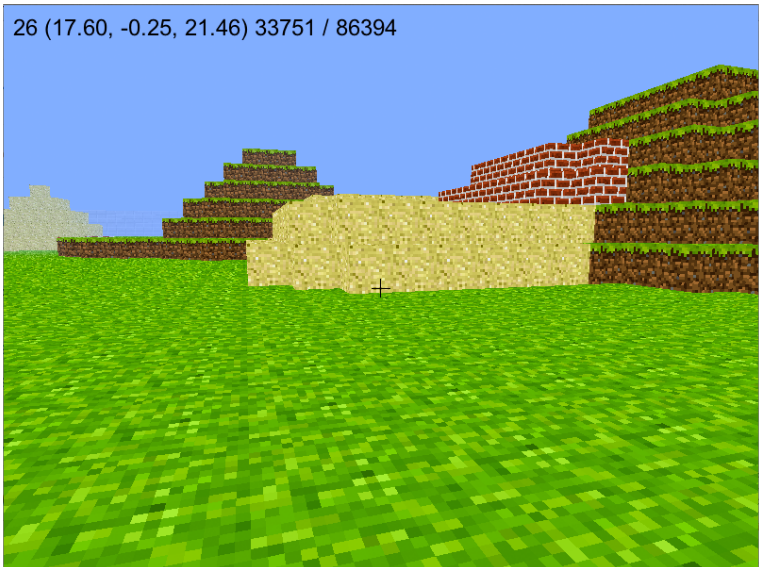

Figure 3.7: *A screenshot of* `Minecraft` *by Michael Fogleman*

3.6 Summary

An overview of Minecraft, its history and its basic concepts, as well as other software in its periphery, have been presented in this chapter. Minecraft has been chosen to become a simulation environment for MicroPsi because it is comparably easy to build custom client software that can perform all the actions on a Minecraft server that a human player could. Minecraft is a complex, yet easily accessible virtual world, that shows characteristics of real environments. It is in constant development and new features are added regularly. It has a massive fanbase and a huge community around all kinds of game modifications. The most important concepts of Minecraft in respect to this thesis are blocks, chunks and also the client server protocol. They will be used for the implementations described in chapter 4.

As a basis for the implementation of the Minecraft interface for MicroPsi, the bot framework `Spock` by Nick Gamberini has been chosen to extend MicroPsi. Furthermore, `Minecraft` by Michael Fogleman has been chosen to serve as a foundation for the visualisation component. Their implementations will be presented in detail in the following chapter as well.

[7] `http://www.reddit.com/r/programming/comments/1b8a6z/simple_minecraft_clone_in_580_lines_of_python/`

[8] `https://news.ycombinator.com/item?id=5458986`

[9] `http://www.pyglet.org/`

Chapter 4

Minecraft as a MicroPsi 2 World

The objective of this project is to build and test an interface in between MicroPsi and Minecraft, so that a Minecraft world can be used as a simulation environment for experiments within the MicroPsi framework, which will act as an artificial player. Since there exist open source projects that perform many of the tasks required for this goal, `Spock`, the Python Minecraft bot framework by Nick Gamberini, and `Minecraft`, the Python Minecraft clone by Michael Fogleman, have been chosen as foundations that have been implemented into the MicroPsi framework. `Spock` facilitates communication with Minecraft servers. To make the new simulation environment monitorable, `Minecraft` by Fogleman has been used to implement a 3D visualisation of the Minecraft world in the web interface. The following section gives an overview of the implemented modules.

4.1 Overview

The modular architecture of MicroPsi allows it to add new simulation environments (or worlds, as they are called in MicroPsi) fairly easily. To communicate with a MicroPsi node net, a world needs an interface, which is called *world adapter*, as it is described in section 2.5. Looking at the Minecraft side, communication with a Minecraft server typically requires a constant flow of data packets going in and out, as it is described in section 3.5. To add a Minecraft world to MicroPsi, the demands of both sides have to be met.

The contributions of this project are divided into the four modules `MinecraftWorld`, `MinecraftWorldadapter`, `MinecraftClient` and `MinecraftVisualisation`. The resulting architecture is displayed in figure 4.1. The `MinecraftClient` manages the communication with the Minecraft server, provides convenient functions and data structures for sending and responding to packets and stores and regularly updates a simple representation of the environment data it receives from the server. The `MinecraftVisualisation` module generates 3D images that display the current state of the Minecraft environment based on the world data it receives from the `MinecraftClient`. The `MinecraftWorldadapter` represents the interface towards the MicroPsi node net. It defines the *data sources* that are filled with data from the `MinecraftClient` and accessed by the node net. It furthermore defines the *data targets* that, conversely, are filled by the node net and translated to actions that the `MinecraftClient` executes. What ties it all together is the `MinecraftWorld` module. It provides a step function that is called frequently by the world runner and that advances the `MinecraftClient`, the `MinecraftWorldadapter` and the `MinecraftVisualisation`.

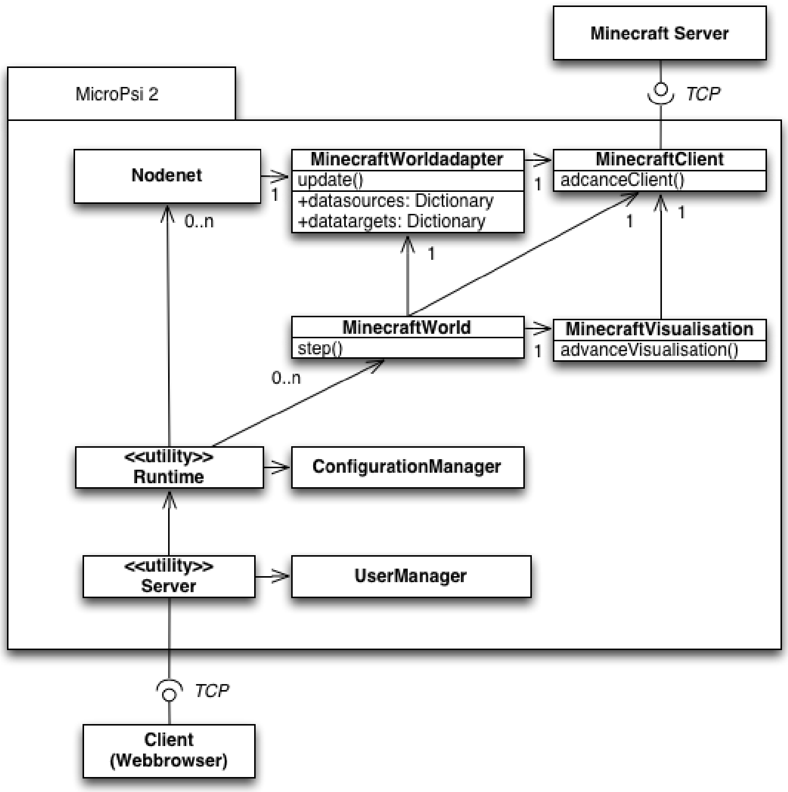

Figure 4.1: *The architecture of MicroPsi with the new Minecraft interface*

The `MinecraftVisualisation` module can be exchanged or cut off completely very easily, as no other modules depend on it. Instead of the visualisation, a placeholder image can be displayed in the web interface, which does not effect the functionality of the simulation. The `MinecraftVisualisation` module itself depends on the data structures of the `MinecraftClient`. This means, exchanging the `MinecraftClient` would require adjustments of the `MinecraftVisualisation`, to still function as intended. The same holds true for the *data sources* and *targets* in the `MinecraftWorldadapter`.

4.2 Using `Spock` as the `MinecraftClient`

As mentioned above, the purpose of the `MinecraftClient` is to manage the communication with the Minecraft server and to provide a representation of the agent's environment. The calculation of the simulation environment, does not take place in MicroPsi itself, but on a regular Minecraft Server. Instead, `Spock` is integrated into MicroPsi and represents the simulation world towards it. `Spock` (in the following the `MinecraftClient`) communicates with the Minecraft server via the *client server protocol* and provides data that can be used as *data sources* for the world adapter and translates the data from the *data targets* to actions in the simulation environment. That way, to MicroPsi it looks like the `MinecraftClient` is the simulation environment itself, where, in fact, it is the interface to the game world server.

The original event loop of the bot framework had to be dissolved and rebuilt as an `advanceClient` function that is called as a part of the `MinecraftWorld`'s step function. For every invocation of the `advanceClient()` function the `MinecraftClient` sends its packet queue to the server through a TCP socket, receives incoming packets from the server, dispatches the received packages appropriately and, if necessary, prepares the necessary actions that it is instructed to by the `MinecraftWorldadapter`'s *data targets* (see figure 4.2).

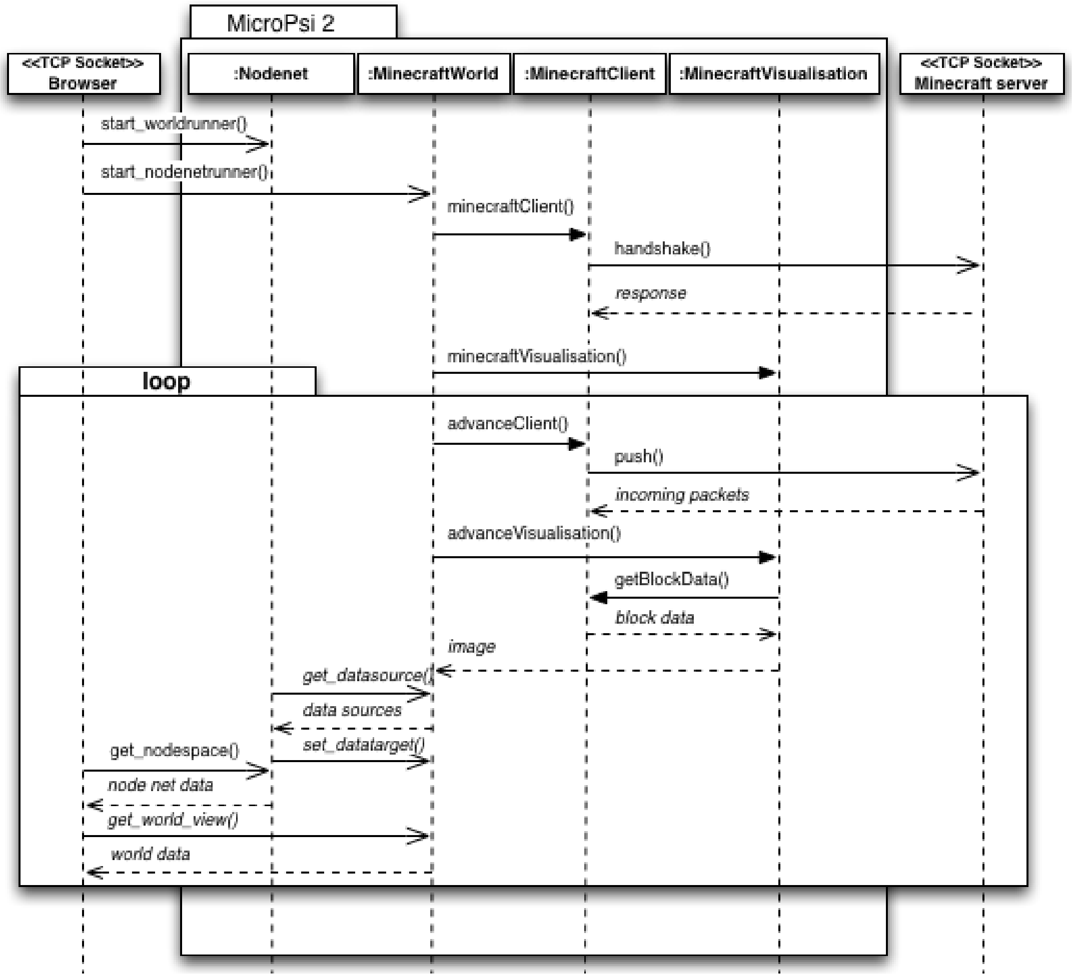

Figure 4.2: *A simplified sequence diagram of the flow of data in between the web browser, MicroPsi and a Minecraft server in respect to the contributions of this thesis*

Subsequently, a system for translating *data targets* to actions had to be implemented. In most cases, performing actions means to let the `MinecraftClient` send a particular set of packets to the Minecraft server. For this purpose, the class `PsiDispatcher` has been added to the `MinecraftClient`, which is described in section 4.2.2.

4.2.1 Overview of the `MinecraftClient`

The `MinecraftClient` holds all the functionalities that enable it to act as an artificial Minecraft player. It implements the packet based *client server protocol* and uses it to communicate with Minecraft servers. It provides data structures that reflect the concepts of a Minecraft world and uses these to store the data it receives from the server. It is heavily based on `Spock`, which has been developed as an educational project. The scope of the original project was to build a pure Python Minecraft bot framework without dependencies. This has been achieved with one minor exception:

if one would like to connect the bot to an official Minecraft online server, the packets have to be encrypted using the Python cryptography library PyCrypto. It consists of several classes, which are outlined in figure 4.3.

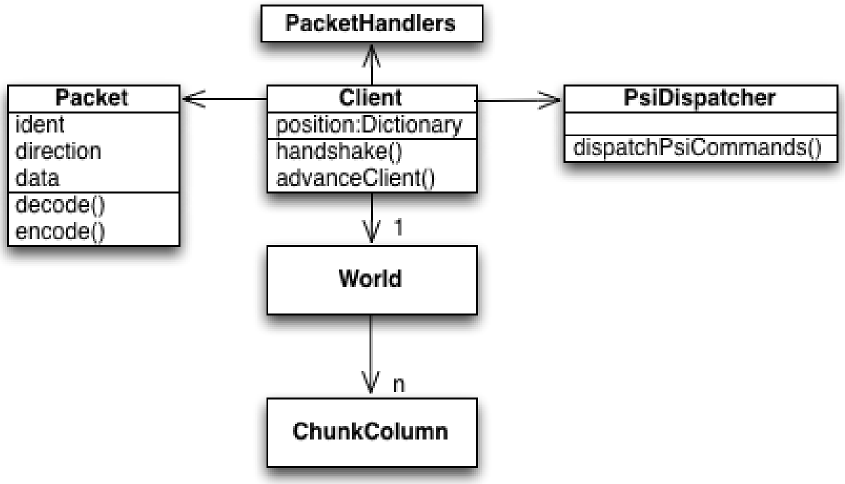

Figure 4.3: *The most important classes of the `MinecraftClient`*

The main class `Client` provides the internal logic that is necessary to implement communication with Minecraft servers. Most importantly, these are the function handshake(), which initiates the communication with the server by performing the actions of the *login sequence*[1] and the function advanceClient() that performs one iteration of the client's lifecycle.

The class `Client` furthermore holds a reference to an instance of the class `World` and uses instances of the classes `Packet` to implement the *client server protocol*. The class `World` holds the internal representation of the gameworld. It brings functions, data structures and classes that represent single sections of the environment (such as `ChunkColumns`) to represent the gameworld and to deliver information about what block sits where. `ChunkColumns` manage dictionaries that store chunk data as integers.

Instances of the class `Packet` represent single packets of the *client server protocol* as they are described in section 3.2.2. They bring functions to encode and decode packet data to get to the payload or to be able to send packets to the Server.

The collection of classes `PacketHandlers` registers one handler class for each packet type that the client is supposed to deal with by default. For every iteration of the `advanceClient()` function incoming packets are dispatched to these handlers. Figure 4.4 gives an example.

4.2.2 Extensions to the `MinecraftClient`

To function as a simulation world for MicroPsi, the `MinecraftClient` has to translate activation of the *data targets* in the `MinecraftWorldadapter` to actions. Therefore, a reference to the new class `PsiDispatcher` has been added to the `MinecraftClient` that holds a single function `dispatchPsiCommands()`. Its purpose is to check the values of the *data targets* frequently and invoke appropriate actions, if necessary. The `PsiDispatcher` is called as a part of the `advanceClient` function. It checks each

[1] `http://wiki.vg/Protocol_FAQ`

```
#Chunk Data - Update client World state
@phandle(0x33)
class handle33(BaseHandle):
   @classmethod
   def ToClient(self, client, packet):
      client.world.unpack_column(packet)
```

Figure 4.4: *By default, this function handles "Chunk Data (0x33) packets.*

available data target, and if necessary invokes an appropriate action (eg. sending a packet). The following listing gives an example for a data target that has been specified to indicate that the agent is supposed to move one block towards the direction of the x-axis.

```
#check for MicroPsi input
if self.client.move_x > 0:
    self.client.push(Packet(ident = 0x0B, data = {
        'x': self.client.position['x'] + 1,
        'y': self.client.position['y'],
        'z': self.client.position['z']
        'on_ground': False,
        'stance': self.client.position['y'] + 0.11
        }))
```

Figure 4.5: *The `PsiDispatcher` checking for activation of the move-x data target and invoking the according action by sending a "Player Position" packet*

4.3 Module `MinecraftVisualisation`

Another important part of this project is the visualisation component. There are two main reasons for its realisation. The first reason is that the agent's behaviour within the simulation environment is supposed to be visually monitored from the MicroPsi web interface—in a both efficient and pleasurable manner. The second reason is that the image data is supposed to be processed by the node net as a data source itself in the future. The module contains classes and functions that provide an interface to the OpenGL context that generates the 3D graphics. This section gives an overview about what data the visualisation uses to generate images of the environment.

Inside the `MinecraftWorld`'s step function, the visualisation module is called to generate a 3D model of the Minecraft world and the agent within. The visualisation component reads from the `MinecraftClient`'s internal gameworld representation to generate the 3D model. This means that, from pure Minecraft world data, a 3D visualisation is generated. It is supposed to display a bird's-eye view of the current chunk to give a good overview of the bots environment and forward it to the web interface.

Specifically, the representation of the chunk, the agent is located in, is fetched and for each solid cube in this chunk, a corresponding cube is rendered within the visualisation using Pyglet's OpenGL abstraction (see figure 4.6). Each block has textures matching its type. The implemented format for the textures is compatible to the widely available Minecraft texture packs. That way, the visualisation's look can be changed

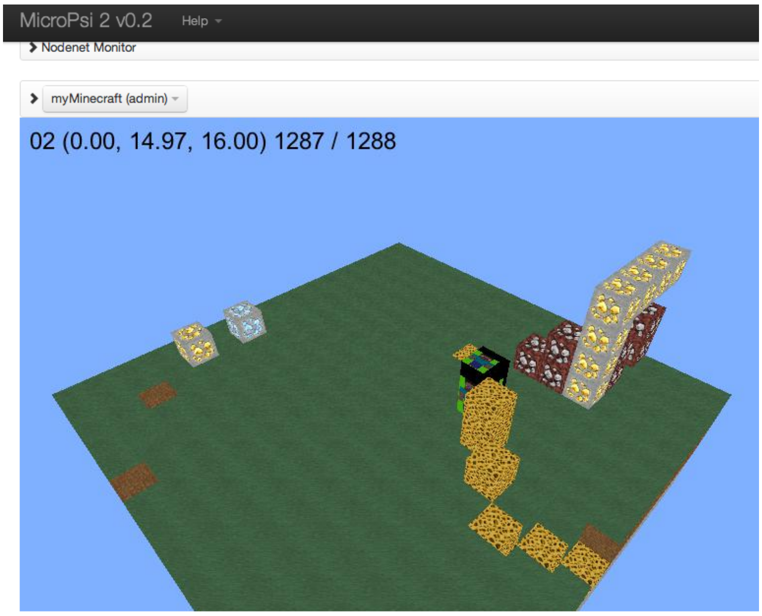

Figure 4.6: *An image generated by the `MinecraftVisualisation` component*

completely within seconds. The resulting images are exported as PNG files which are stored in memory and which are delivered to the web interface which is requesting them frequently. A refresh rate of six or more images per second creates the impression of a video stream.

The resulting architecture can be described as an implementation of a *model-view-controller* pattern (see figure 4.7). Therefore the `MinecraftWorld` acts as the *controller* that iterates through the life cycles of both the `MinecraftClient` and the `MinecraftVisualisation`. Everytime the visualisation component that resembles the *view* updates the 3D view, it fetches the required world data from the `MinecraftClient`, which acts as the *model*.

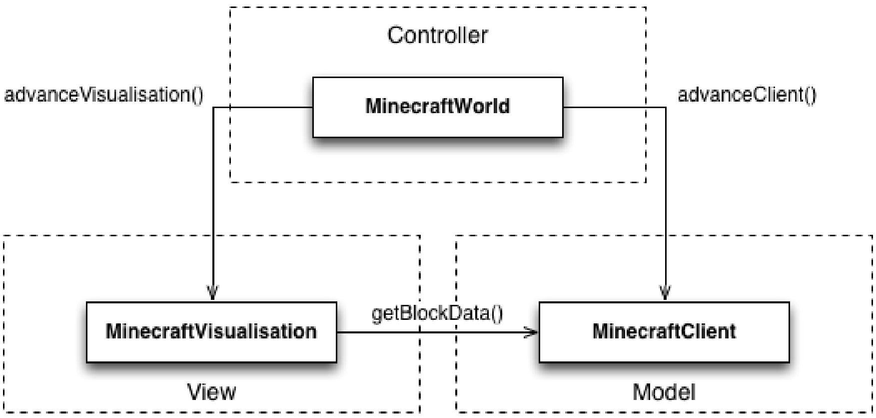

Figure 4.7: *The model-view-controller pattern used for the visualisation module*

4.4 Module `MinecraftWorldadapter`

The `MinecraftWorldadapter` provides the necessary interfaces that let a `MinecraftClient` be controlled by a MicroPsi node net. Most importantly, these are the before mentioned *data sources* and *targets*. It inherits mutator methods for these from its parent class `WorldAdapter`, which itself inherits from `WorldObject`, which is the superclass of all agents and world objects, as they are described in section 2.5.3.2. As it holds a reference to the according `MinecraftClient`, it has full access to the environment data and other properties of the client. This allows it, to easily implement simple "sensors" by developing algorithms that utilise and search the provided data structures. An example is given in the following section.

4.5 Case Study

To explain, how the new interface can be utilised, we will construct a simple case study in the following. As it only consists of a primitive wiring between sensors and actuators, the scientific value in terms of AI research is low. However, it shall suffice as a proof of concept that the interface is functional and may serve as a foundation for future experiments.

4.5.1 Implementation

The scenario consists of a single agent that is placed in a Minecraft world. It is standing on a flat surface that is made out of *grass* blocks. Next, a block of *diamond* is placed in the same chunk. The agent is supposed to automatically approach this block, depending on where it has been placed. Therefore it is controlled by a MicroPsi node net, which is set up as described in the following.

Node net

After the web interface has been started, a new node net is generated. Inside it, four sensor nodes have been created. These nodes shall represent sensors, that can detect if a block of diamond can be located in a particular cardinal direction. Furthermore, four actuator nodes are created, that represent the agent's movements towards each of the same four directions. Next, four links are generated that in the following forward activation from the sensor nodes to the actuator nodes that make the agent move towards the according direction. Finally, an activator node is added, that defines the general ability of links in this particular node net to forward activity. The resulting node net is displayed in figure 4.8.

Data sources

To implement the before mentioned sensors, suitable algorithms have to be integrated into the `MinecraftWorldadapter`. For sensors that detect *diamond* blocks in the current chunk, this could be realised by fetching the chunk data and examine it for the according block type. If a block is found, its coordinates are modified in respect of the agent's own position to express the sensor values. We furthermore define that the activation of the sensors equals the distance to the *diamond block* in the regarding direction. Additionally, we define a threshold of a distance of two blocks. If the agent

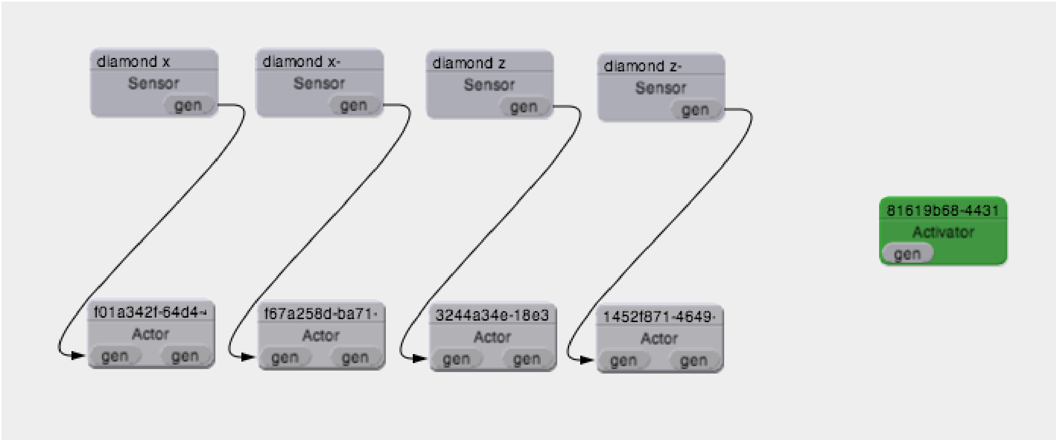

Figure 4.8: *The node net set up for this experiment*

is closer to the target then this threshold, the sensors will stop sending activation. A possible implementation is outlined in figure 4.9.

```
for y in range(0, 16):
 current_section = current_column.chunks[int((bot_block[1] + y -  2) / 16)]
 if current_section != None:
  for x in range(0, 16):
   for z in range(0, 16):
    current_block = current_section[’block_data’].get(x, _block[1] + y - 10 / 2)), z)
    if current_block == 56: #Diamond Ore
     diamond_coords = (x + x_chunk * 16,y,z + z_chunk * 16)
     self.datasources[’diamond_offset_x’] = bot_block[0] - coords[0] - 2
     self.datasources[’diamond_offset_z’] = bot_block[2] - coords[2] - 2
     self.datasources[’diamond_offset_x_’] = (bot_block[0] - coords[0]) * -1 - 2
     self.datasources[’diamond_offset_z_’] = (bot_block[2] - coords[2]) * -1 - 2
```

Figure 4.9: *Searching the current section for a diamond block and, if it is found, filling the concerning data sources with the distance towards it.*

Next, in the node net editor the four sensor nodes are edited in such way, that they are associated with the new *data sources*. For every iteration, the node net will now update the sensor values with the values calculated for the *data sources* of the `MinecraftWorldadapter`.

Data targets

Subsequently, data targets have to be defined, that represent the actuators for moving forwards, backwards, left and right. Moreover, suitable methods have to be integrated into the `PsiDispatcher`. The example from section 4.2.2 illustrates just this. Equally, in the node net editor the actuator nodes have to be associated with the new *data targets*.

Execution

To conclude the experiment, a chunk in a Minecraft world is set up in a way, that it contains only a flat surface, without other obstacles, that the agent might move around on freely, as well as a diamond block that is placed at its center. Then, the agent is placed in one of the chunk‘s corners.

Next, the *world runner* and the *node net runner* are started through the web interface. For every step of the world, the sensor nodes are being filled with the sensor values and forward the activation to the actuator nodes. Activated sensor and actuator nodes light up green. The client checks the data targets and therefore moves towards the diamond. Once it gets to a distance towards the diamond, that is below the defined threshold, the sensors stop sending activation, and the agent stops moving towards it. Screenshots of the simulation run can be found in figure 4.10. Note, that if the node net runner and the world runner are stepped asynchronously, there might be a delay in the shift of behaviour of the agent. Hence, it is advised to run them synchronously or at least with a timing close to each other.

4.5.2 Evaluation

Now, that a test run has been concluded, the implementation can be evaluated. The experiment shows that a functional interface in between a Minecraft world and the MicroPsi framework has been implemented. In the experiment, the agent concludes the appropriate action, if a node net actuator node receives activation and stops doing so, when it stops receiving activation.

Surprisingly, even on a reasonably up-to-date computer[2], the simulation runs comparably slow, whenever the visualisation component is activated. Where without it, 15 to 30 iterations of the simulation per second can easily be performed, with an active visualisation component, hardly 4 to 6 iterations can be achieved. The bottleneck presumably is that with the current implementation each image has to be transferred via its own HTTP request. The simulation should assumedly speed up if the image data would be transported through a web socket, for example.

With MicroPsi node nets in the back, more complex experiments can now be thought of and implemented, that lead to more complex behaviour of the agent. Therefore, a basic understanding of Python and the code at hand as well as the *client server protocol* is required to implement new data sources and targets. An alternative could be that along with the MicroPsi framework, a set of predefined *data sources* and *targets* that represent the most common forms of interaction with the game world could be delivered. This would make it possible, to develop more sophisticated experiments without touching the Python code. Instead, only the graphical editor would be used to assemble node nets that lead to behaviour more interesting to study.

4.6 Summary

This chapter presented the implementation of the Minecraft interface for MicroPsi, which consists of the modules `MinecraftClient`, `MinecraftVisualisation` and `MinecraftWorldadapter`. The `MinecraftClient` facilitates the communication with the Minecraft server; the `MinecraftVisualisation` delivers OpenGL rendered image data of the environment and the `MinecraftWorldadapter` serves as the actual interface in between the `MinecraftClient` and a MicroPsi node net. Next, a case study has been performed and evaluated. A MicroPsi client had to move towards a block of a particular type. It showed that the

[2]2.26 GHz Intel Core 2 Duo, 8GB DDR3 RAM, NVIDIA GeForce 9400M 256 MB, OS X 10.8.3 , Chrome Version 29.0.1547.65

interface is functional and ready for more sophisticated experiments. With the implementation of the interface and the following proof-of-concept case study the scope of this thesis has been completed. The last chapter contains a summary and an outlook towards future applications.

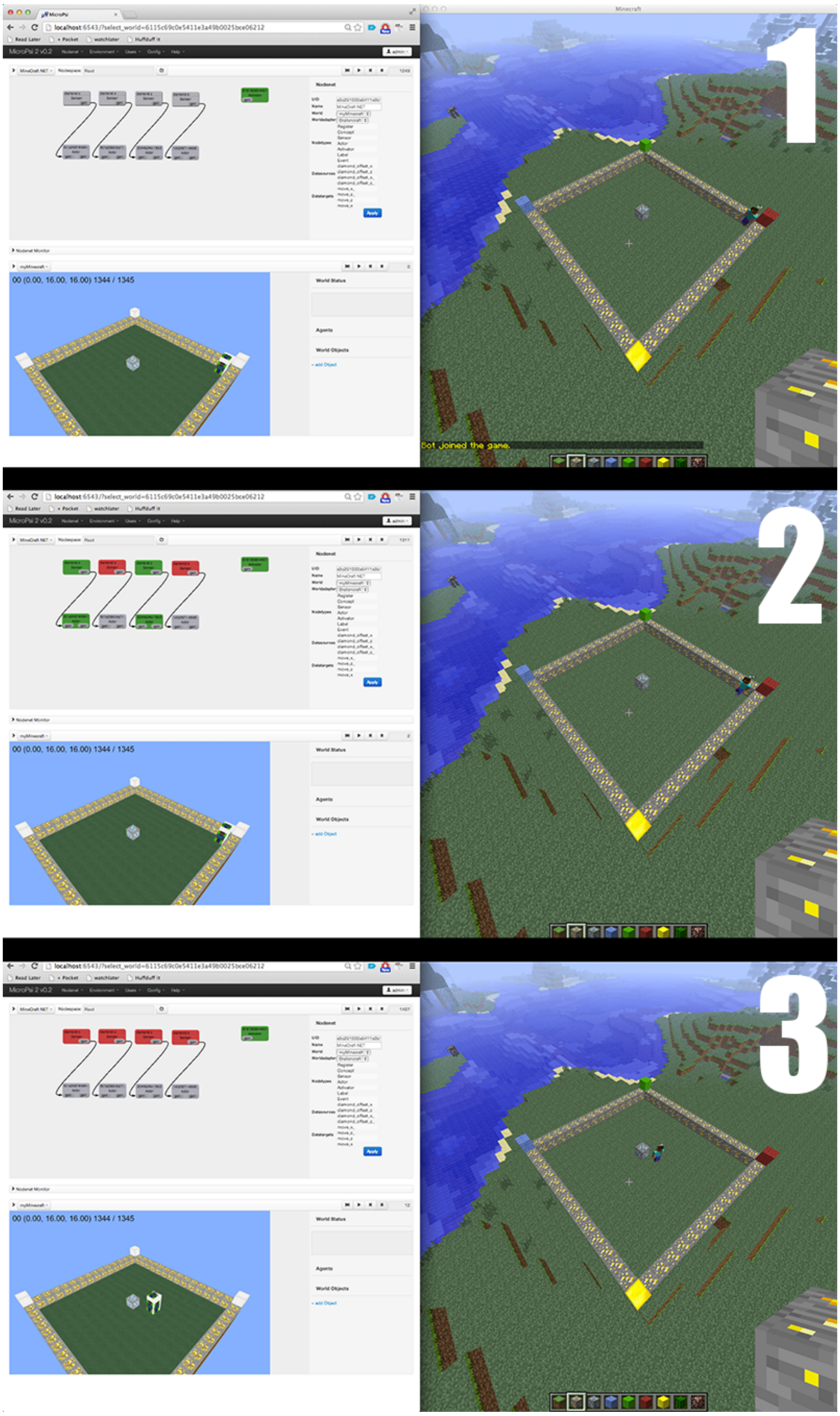

Figure 4.10: *1. Neither node net, nor agent show any activity, as the experiment did not start yet. 2. The sensors that indicate a diamond towards the negative x- and z-axes light up green—so do the according actuators. The agent starts moving. 3. The agent arrived close enough to the diamond that the sensors stop forwarding activation. The agent stops moving.*

Chapter 5

Conclusion

This thesis presented an implementation of an interface that makes it possible to use servers of the video game Minecraft as a simulation environment for the cognitive artificial intelligence framework MicroPsi 2. After several iterations and trying out different approaches and technologies, the interface is now functional. The proof-of-concept experiment with an agent seeking a particular block resulted in proofing that Minecraft is usable as a simulation environment.

In particular, an artificial player may now connect to a Minecraft world and be controlled by a MicroPsi node net. Because there is a wide variety of possible input data from Minecraft worlds and of possible actions artificial players can perform, corresponding data sources and targets may furthermore be defined freely inside the world adapter code.

The interface is in its core based on two open source software projects. The Python Minecraft bot framework `Spock` serves as the client that connects the architecture to Minecraft servers. The Python Minecraft clone `Minecraft` by Michael Fogleman serves as the foundation for the visualisation component.

The implemented case study shows that the interface as a whole works as expected. Because it so far serves foremost as a proof-of-concept, there is a lot of room left for improvement. The two main critical aspects are speed and robustness. Speeding the simulation up would require more profiling and finding bottle necks in the data flow. Most likely, at this time one of the major bottlenecks will be that for every iteration of the visualisation component an image file has to be transferred through a HTTP connection.

The Minecraft protocol proved to be easily accessible, understandable and therefore gratifying to work with. It sets the hurdles for programming against Minecraft very low, so there might even be research projects with other scopes than AI simulation that could make use of Minecraft infrastructures.

Possible future applications of this project would be implementing data sources and targets for every aspect of a Minecraft world and for every action that one wants to experiment with. For example, these could include interacting with the environment by picking up objects, creating and destroying blocks, building, mining, interacting and fighting with other objects. Especially, in the future, multiple agents shall interact within the same environment and collaborate with each other.

A complex experiment that would make use of the unique capabilities of MicroPsi and Minecraft could look like the following: A group of Psi agent get placed in a Minecraft world. They are equipped with a set of predefined drives. These could in the most simple case be the need to consume specific resources like water and food at least once a day. As they would not know about these drives themselves, they would start exploring their environment randomly. Soon they would learn, what actions lead to pleasure and what to displeasure signals. Additionally they would start learning the semantics of the game world to recognise related objects, even though they look different. Ideally, the agents would communicate their knowledge in between each other and therefore help each other to survive longer. In an identical world human players could be given the same task. Ultimately, their behaviours could be compared to find out, if the Psi implementation shows signs of the ability to recreate human cognition.

List of Figures

List of Tables

Contents of the CD

The attached CD contains the following::

- this bachelor thesis in PDF format,
- the Python project MicroPsi including the Minecraft world adapter,
- the binaries of a Minecraft server that can be used to test the implementation.

Bibliography

[Bac09] Joscha Bach. *Principles of Synthetic Intelligence PSI: An Architecture of Motivated Cognition.* Oxford University Press, Inc., New York, NY, USA, 1st edition, 2009.

[Bac12] Joscha Bach. Micropsi 2: The next generation of the micropsi framework. In Joscha Bach, Ben Goertzel, and Matthew Iklé, editors, *AGI*, volume 7716 of *Lecture Notes in Computer Science*, pages 11–20. Springer, 2012.

[BB] Gergo Balogh and Arpád Beszédes. Codemetropolis–a minecraft based collaboration tool for developers.

[DaS02] Dietrich Dörner and Christina artl Storck. *Die Mechanik des eelenwagens.* Huber, 2002.

[Doe]

[Dun11] Sean C. Duncan. Minecraft, beyond construction and survival. *Well Played*, 1(1):1–22, January 2011.

[GLA+10] Ben Goertzel, Ruiting Lian, Itamar Arel, Hugo de Garis, and Shuo Chen. A world survey of artificial brain projects, part ii: Biologically inspired cognitive architectures. *Neurocomputing*, 74(1–3):30 – 49, 2010. ¡ce:title¿Artificial Brains¡/ce:title¿.

[GP07] Ben Goertzel and Cassio Pennachin. *Artificial general intelligence.* Springer, 2007.

[ima13] August 2013.

[mcw13] August 2013.

[pro13] August 2013.

[Rus09] Stuart Russell. Artificial intelligence: A modern approach author: Stuart russell, peter norvig, publisher: Prentice hall pa. 2009.

[sky13] August 2013.

[Squ07] Kurt Squire. Open-ended video games: A model for developing learning for the interactive age. In *The Ecology of Games: Connecting Youth, Games, and Learning*, pages 167–198. The John D. and Catherine T. MacArthur Foundation Series on Digital Media and Learning, 2007.