Institut für Informatik
Lehrstuhl für Programmierung und Softwaretechnik

LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Minecraft in MicroPsi

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

Ziel der Arbeit ist die Entwicklung und das Testen einer Simulationsumgebung für einen kognitiven Agenten auf Basis des populären Videospiels Minecraft.

Minecraft bietet sich als Grundlage für eine Simulationsumgebung aufgrund der kompositionalen Semantik der Spielwelt besonders an, da das Agentensystem durch das Erforschen seiner Umwelt Wissen über die Spielwelt aufbauen und ähnliche Strukturen wieder erkennen kann. Objekte der Spielwelt sind in Minecraft keine bloßen Hindernisse sondern werden mit variablen Eigenschaften prozedural erzeugt und ähneln so eher einer realen Umgebung als andere virtuelle Welten.

Da Minecraft von Anfang an mit einem Mehrspielermodus ausgestattet wurde, lässt es sich zudem für Multiagenten-Umgebungen und so für kollaborative Agenten verwenden. Des Weiteren 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.

Angestrebt wird, dass die kognitive Architektur MicroPsi2 sich an einen Minecraft-Server anmelden kann, ihre Umgebung wenigstens in Ansätzen wahrnimmt (Objekte, Terraintypen), sich fortbewegen kann und einfache Interaktionsmöglichkeiten besitzt (z.B. Objekt aufnehmen und ablegen).

Daneben soll eine Visualisierung der Umgebung aus Sicht des Agenten entstehen, z.B. als zweidimensionale Übersicht, in deren Zentrum der Agent steht. Diese Visualisierung soll live im Browser angezeigt werden (wahlweise mit WebGL oder Canvas/D3).

Zum Schluss der Arbeit soll zum Testen der Funktionalität ein virtuelles Braitenbergvehikel in die Simulationsumgebung gesetzt werden, welches sich daraufhin auf die nächstgelegene Lichtquelle zubewegen soll. Dieses Experiment wird als Teil der Arbeit umfangreich dokumentiert.

Abstract

What I cannot create, I do not understand Richard P. Feynman, 1988

Danksagung

Ich danke meinen Betreuern Joscha Bach und Annabelle Klarl sowie Dominik und Professor Wirsing. Nick Gamberini, Fogleman

Contents

Chapter 1

Introduction

The hunt for artificial intelligence started many years ago. Dividing the subject into the strong and the weak part means separating its goals into useful applications and those that try to learn about the nature of intelligence itself. The ultimate task of strong A.I. — recreating human intelligence — admittedly still seems to be science fiction, though.

But then again: a new generation of cognitive scientists, psychologists and computer scientists strives to implement new ideas for simulated cognition, by building cognitive architectures. Many of them do so by simulating, in one way or the other, what could be called neural node nets.

One of these architectures is MicroPsi 2. Developed by Joscha Bach and Dominik Welland, it is based on the Psi Theory by Dietrich Dörner, who is a german Professor for theoretical psychology. It aims at providing a solid and complete implementation of his theory of cognition and at the same time being easily accessible, understandable and modifiable for the research and applications to come.

To test the functionality of such cognitive architectures and to figure out their capabilities and potential, we need to research their behaviour inside defined environments. As implementing AI into the physical world (as robots for examples) requires building appropriate hardware and patience, computer-simulated environments therefore play an important role.

MicroPsi 2 is both: a cognitive architecture but also a set of and an interface to simulation environments.

1.1 Motivation

Video games are natural applications of artificial intelligence. The quality of a games A.I. can make all the difference in between a great title and an uninspired demo of computer graphics technology. One game that especially stood out in the past is Minecraft. As a so called sandbox game, it attempts to give the player as few limitations as possible for what they do in the game world. This kind of relatively open environment, delivers a a simulated world rich of opportunities, to both human and artificial players. 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, with the most important one on the AI side being MicroPsi 2, the most ambitious 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 creation as well as explanations of their basic ideas and relevant insights to their architecture are what this chapter is about.

Chapter two and three give an overview of the necessary foundations for this project. Chapter two introduces the Psi theory and its implementations. We will especially focus on the implementation MicroPsi 2 and explain it in detail Chapter three introduces the video game Minecraft, its protocol and related software projects. As the resulting software heavily relies on them, the protocol and the utilized open source software will get extra attention. In Chapter four the contribution of this thesis — the Minecraft world adapter for MicroPsi — is described in detail. As a proof of concept, the description eventually leads to a small experiment. Chapter five summarises the findings of this thesis and gives an outlook on possible future applications.

Chapter 2

Psi

A widely accepted definition of the field of AI is that it is the study and design of intelligent agents, where an intelligent agent is a system that perceives its environment and takes actions that maximise its chances of success.

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, the game show, that is 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. For AI in scenarios of this kind, the term applied AI (or weak AI) has been coined.

Strong AI, in contrast, is about researching the nature of intelligence itself. An actual (hypothetical) implementation of a Strong AI translates to building a machine, that is capable of acting like a human being — not just in a defined problem fields, but in all of them.

Cognitive AI, in particular, can be thought of as architectures that implement findings and theories in the fields cognitive science and psychology, as well as the neuro-sciences, for the sake of proving, if the theories hold against what they promise.

Many cognitive architectures share characteristics with or directly implement artificial neural node nets.

2.1 Psi Theory

The Psi theory in its foundations was described by german psychologist Dietrich Dörner in his books "Bauplan für eine Seele" and "Die Mechanik des Seelenwagens" from 1998 and 2002. Dörners holistic approach goes beyond classic problem solving but develops a unified model for cognition that implements motivation and emotions. 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 eventually lead to a deeper understanding of cognition itself.

It's main ideas are, that it thinks of cognition as a graph-like structure (e.g. node-net) of relationships that strives to maintain homeostatic balance.[Bac09]

Basic components of the theory are Representation, Memory, Perception, Drives, Cognitive modulation and emotion, Motivation, Learning and Problem solving as well as Language and consciousness.

"The basic conceptual element, analogous to Dietrich Dörner's Psi theory, is the Quad. It makes use of a single 'gen' slot and the four directional gates 'por', 'ret', 'sub', 'sur'. 'Por' encodes succession, 'ret' predecession, 'sub' a part-of relationship, and 'sur' stands for has-part. With the 'gen' gate, associative relationships can be expressed."

Joscha Bach adapted that theory to bring it in a contemporary from with slight modifications in "Principles of Synthetic Intelligence PSI: An Architecture of Motivated Cognition" [Bac09].

Even though building a conscious machine that thinks and acts like we do is still mere science-fiction, it is this kind of foundational research, that leads no new ways of thinking of the world, that give us our most import leaps.

2.2 Psi Implementations

Psi has been implemented by different groups at different times. The first implementations are by Dörner and his associates themselves (see figure 2.1). They used Pascal and developed it for Windows environments. This implementation can still be downloaded and runs on Windows 7 installations, for example.

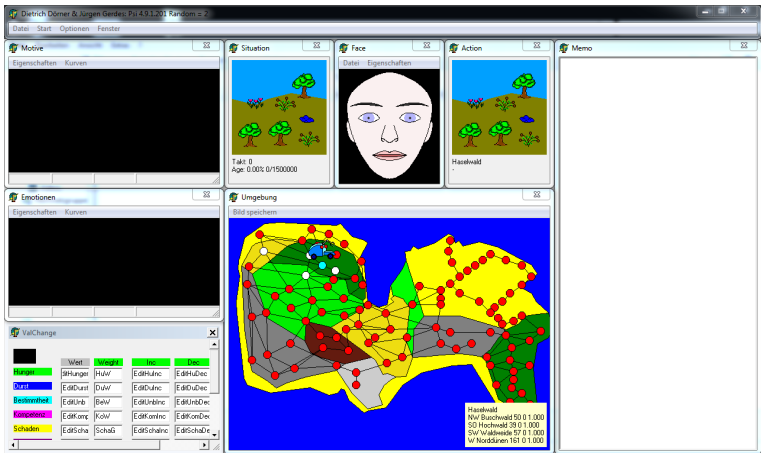

Figure 2.1: *Dörner's Pascal Psi Implementation*

The work on Dörner's team's implementation has not been continued, so Joscha Bach and his associates built 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 included a graphical editor and a 3D simulation-environment.

2.3 MicroPsi 2

Aiming at better 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 webapplication-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 has only two fairly simple ones: a 2D-Island and a map of the public transportation system of Berlin (see figure 2.2). Instead of building a new 3D-world, with this project we set out for something more experimental. More on this in chapter three and four.

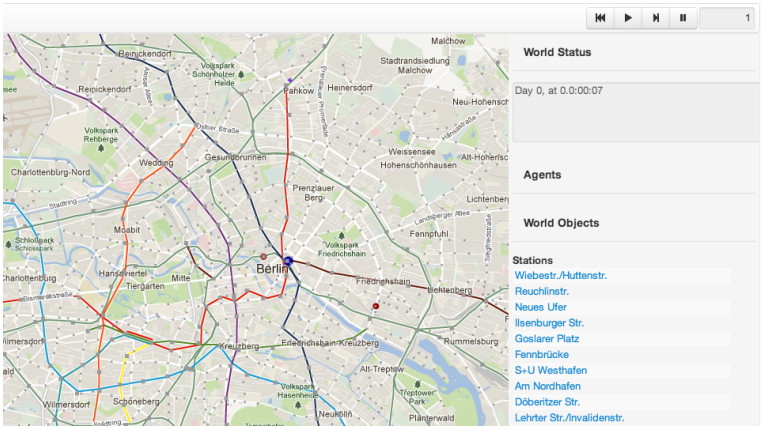

Figure 2.2: *MicroPsi simulation environment Berlin*

2.3.1 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 2.3. First, one can differentiate in between the `Server` module (or the web-interface) and the actual simulation `Runtime` module (also called "core"). In a regular 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 the node net. As these names suggest, the `Runtime` manages both the simulations 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`. [Bac12]

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

2.3.2 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*.

The web interface is naturally 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* library are in use. The graphic renderings (see figure 2.4) utilise the JavaScript graphics library *PaperJS*.

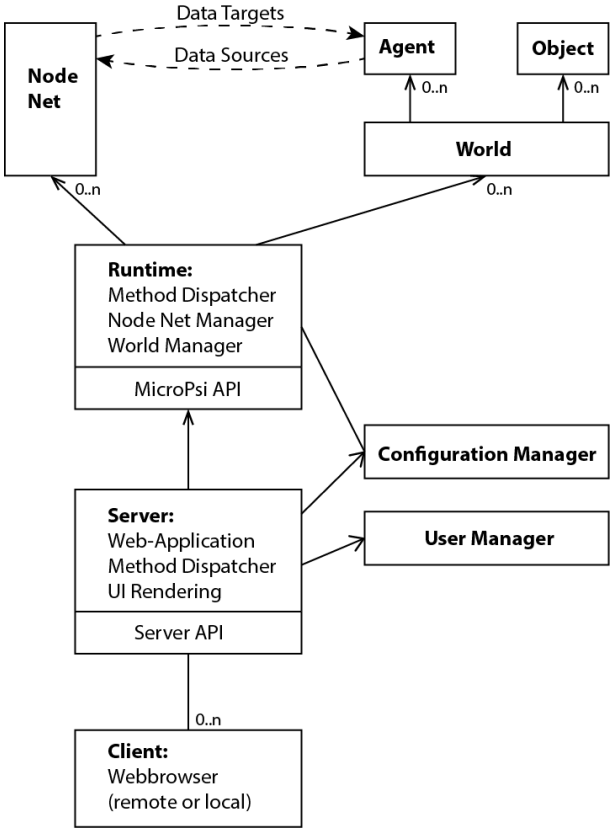

Figure 2.3: *The modular architecture of the MicroPsi framework make it easy to extend. (taken from [Bac12])*

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

2.3.3 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.

2.3.3.1 Node Nets

A MicroPsi node net is defined as a set of states, a starting state, a network function, "that determines how to advance to the next state" and a set of node types. Data sources and data targets serve as input and output towards a world, where a data source is 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.

Nevertheless, nodes may contain 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.

According to the concepts of the Psi Theory, MicroPsi defines agents as node nets

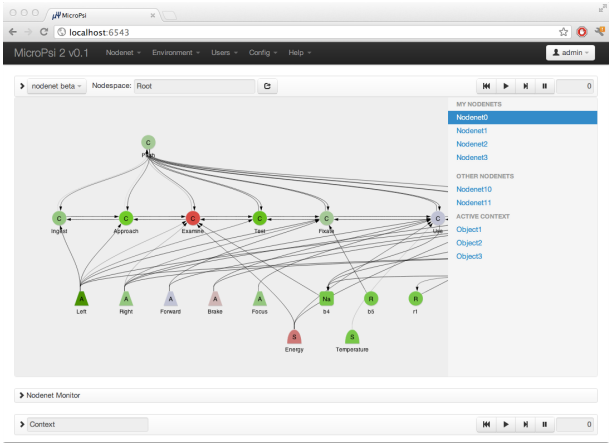

Figure 2.4: *The graphical editor is the primary interface to node nets. (taken from [Bac12])*

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

As node nets share the relevant characteristics with neural node nets, they may enable neural learning paradigms. To store information they can form semantic networks. Furthermore, nodes may contain state machines and other operations, which make it possible to build modularised architectures.

2.3.3.2 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 motoric output. Sophisticated interconnection of those enables interaction with the environment.

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 framework, the priorised 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 functions 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 agents body. The function that advances the object checks for input from the node net and outputs to it.

Chapter 3

Minecraft

The story of Minecraft has many interesting aspects, but first and foremost it is a story of immense, unexpected success.

When Markus ("Notch") Persson built and released the first public version of Minecraft, it soon became clear that his creation resonated with many people. The simple concept of a world entirely build out 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 it's possibilities and boundaries.

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, different versions of the game sold more than 26 million times — with the PC version priced at about 20 Euros, 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 via their own website.

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.

Minecraft is a complex, yet easily accessible virtual world. 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.

Another interesting aspect about Minecraft is the procedural semantic 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 particular characteristics of each tree are generated randomly. This makes a Minecraft world somewhat more realistic than those of many other videogames.

"Minecraft [8] is a popular role-playing game. The game itself does not have a strict game-flow. Its main focus is creativity and the joy of creation. Only the available computation power and the storage capacity can limit the fantasy of the player. The main concept in the game is the block. It is a box with about one meter long sides, compared to the player. Almost everything is built up of it, so the whole World is a 3D matrix filled with blocks of various types. The player can collect the blocks, create (craft) new ones and interact with them. The similar to a virtual Lego with infinite playground and an infinite number of building blocks. Due to its extensibility, its simple yet sophisticated functions, and its rich palette of possibilities Minecraft can

display complex structures with a low overhead "[BB]

3.1 What is a Minecraft world?

In this section basic concepts of the game are described (in regards to our A.I. interface). Minecraft worlds are build out of blocks (see figure 3.1). Blocks are cubes. There are different types (materials) of blocks and they share a single size, which converts to the basic distance unit of Minecraft. One unit can be thought of as roughly equaling one meter.

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 3.1: *A "Grass" Block [ima13]*

Figure 3.2: *A Chunk [ima13]*

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

Figure 3.3: *"The Player" [ima13]*

The game itself has no predefined goals. Players can walk around, discover the generated world (see figure 3.4 1) and collect resources by "destroying" blocks , with the generated resources equaling the block type. They can combine different resources to "craft" items. For example, a player can destroy the blocks that represent a tree (2). The gained "wood" resource could then be used to craft a wooden pickaxe (2), which could then be used to dig into the ground more effectively (4) to "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 (eg. 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 instant access to unlimited resources.

The mode of a Minecraft world does not effect the functionality of this project.

Figure 3.4: *Minecraft Basic Mechanics (CC-BY-3.0 Mojang AB)*

3.2 The Client Server Protocol

Minecraft's Client-Server-Protocol is not publicly documented by the developers themselves. However, the modding-community gathered full 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 with a "Packet ID" byte. The structure of the packet's payload depends on it's Packet ID.

To give an example of one of the easier packets, the "Client Position"-Packet is fairly straight-forward (see figure 3.1). It is exclusively send from clients to servers and starts with it's 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 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: *Structure of the packet "Player Position (0x0B)" [pro13]*

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 of it and send a Client Position packet with these coordinates 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 players 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 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 game console Xbox One is announced. The desktop versions are priced at 19.95 euros, which makes 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 uses, or changes original Minecraft intellectual property. In other words: 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 it's software architecture, data structure and protocols. This is useful for this project, as chances are low that they will have anything against using Minecraft for this project in the foreseeable future.

The Minecraft world with it's logic, semantic and functionalities offers possibilities for an A.I. 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 randomly, 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.

3.4 Related Projects

3.4.1 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" (eg. Minecraft Forge).

One example (and probably the most advanced one), for an entire bot framework that replaces the client, is Mineflayer. [git13] 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.

Opposed to Mineflayer, an example for "game modification" bots are the "Cubebots" — fan-made non-player characters that aim to help Minecraft players with mundane tasks.[mcf13]

3.4.1.1 `Spock` by Nick Gamberini

Developed by Nick Gamberini, `Spock` 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 the existing MicroPsi code and the absence of dependencies (with one exception) leave the code understandable and easy to deploy.

3.4.1.2 Protocol Implementation in `Spock`

The Minecraft protocol implementation in `Spock` is straight-forward (see figure3.5). The necessary data structures are stored separately and can be accessed globally.

The structures are used to parse each packet appropriately (see figure 3.6).

3.4.2 Minecraft Clones

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

Interesting projects include Skycraft [sky13], a Minecraft-like browser-game based on WebGL.

A 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 gained some popularity on reddit [fog13a] and Hacker News. [fog13b]

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 [pyg13].

```
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: *data structures for the packet IDs and structures*

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

        #print hex(self.ident)

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

Figure 3.6: *function for decoding packets*

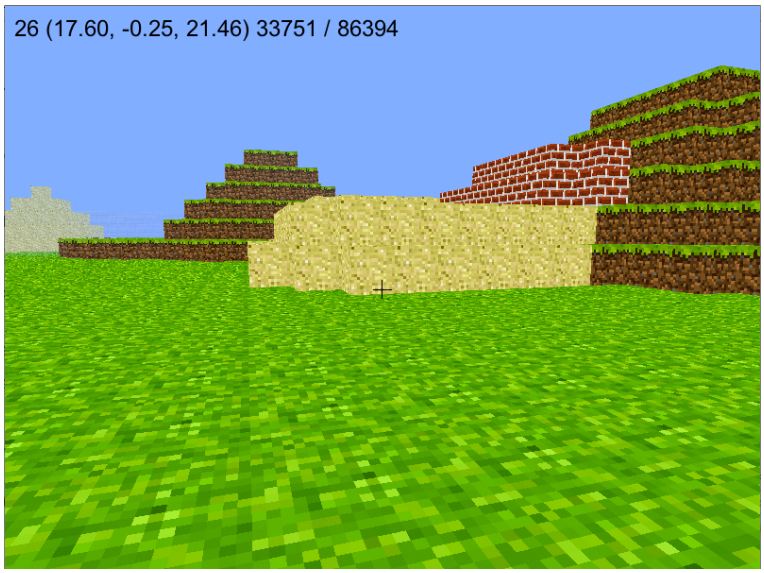

Figure 3.7: `Minecraft` *by Michael Fogleman*

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. To make the new simulation environment monitorable, a visualisation of the Minecraft world has been implemented into 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*. 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.

Looking at the Minecraft side, communication with a Minecraft Server typically requires a constant flow of data packets going in and out. Most third party clients, including Bots, facilitate their own event loops. 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 three modules `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 constantly 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 data it receives from the `minecraftClient`. What ties it all together is the `minecraftWorldadapter`. It provides a step function that advances both the `minecraftClient` and the `minecraftVisualisation` and is called itself by the world runner of the MicroPsi framework. Furthermore it defines and updates the *data sources* and *targets*.

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 webinterface, which does not effect the functionality of sim-

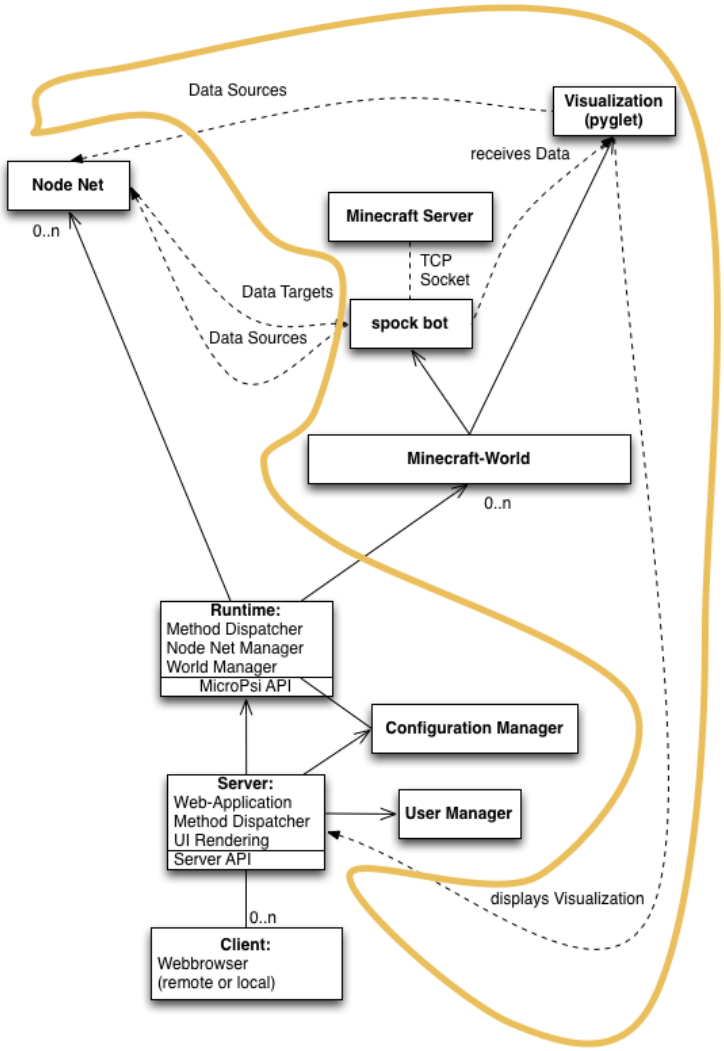

Figure 4.1: *The new architecture of MicroPsi with the Minecraft interface. New modules are framed orange.*

ulation. The `minecraftVisualisation` module itself depends on the data structures of the `minecraftClient` though. This means, exchanging the `minecraftClient` would require adjustments of the `minecraftVisualisation`, to still function as intended. The same holds for the *data sources* and *targets* in the `minecraftWorldadapter`.

4.2 Using `Spock` as the `minecraftClient`

As mentioned before, 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` 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 `Spock` is the simulation environment itself, where in fact it's the interface to the game world server.

The original event loop of the bot framework had to be dissolved and rebuilt as a `advanceClient` function that is called as a part of the world adapter's step function. The event-loop and -handling of Spock had to be slightly modified to work with MicroPsi. It should be noted, that the frequency, with which the framework steps the bot, has to be at least chosen high enough, so that Spock is able to send the necessary keep-alive-signals, to not get kicked from the server.

For every iteration of the event loop, the `minecraftClient` reads incoming data from the socket, dispatches the read packages appropriately and eventually checks the MicroPsi *data targets* to perform an action — if necessary (see figure 4.2). Note, that the described event loop of the original `Spock` is not a loop anymore but each iteration is invoked as a part of the world adapters step function.

Eventually, useful data sources had to be picked and a system of data targets and their translation to actions had to be implemented. In most cases, performing actions means to let spock send a specific set of packets to the Minecraft server.

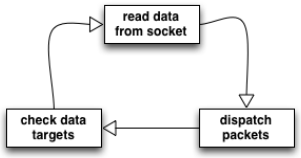

Figure 4.2: *The `minecraftClient`'s event loop*

4.2.1 Overview of the minecraftClient

The `minecraftClient` is heavily based on Spock, the Minecraft bot framework developed by Nick Gamberini as an educational project. It consists of several classes. The main class, `minecraftClient`, holds references to instances of the classes BoundBuffer, World and Packet. Furthermore basic data structures are stored as cflags, mcdata.

The class BoundBuffer is an implementation of a buffer that matches the particular needs of sending and receiving Minecraft packets.

The class World holds the internal representation of the gameworld. It brings functions and datastructures/classes itself to represent chunks and to obtain information about which block sits where.

The class Packet represents a Minecraft packet and brings functions to encode and decode a packet (read from a BoundBuffer) to get to it's payload or to be able to send it to the Server.

The file packet handlers.py provides a class for each packet that the client is supposed to deal with by default. The following listing gives an example.

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

Figure 4.3: *Handling a Chunk Data packet*

The dictionary cflags provides the Socket codes. The file mcdata.py provides dictionaries for the datatypes, blocktype codes, packet names and structures of packets of the Minecraft protocol.

In the file nbt.py several classes for dealing with the NBT file format exist.

The file timer.py contains the classes EventTimer, TickTimer and ThreadedTimer which provide Timing.

The client sets up a socket to the server, starts of with a handshake and then facilitates packet based communication with the server.

4.2.2 Extensions to the `minecraftClient`

It was aimed for to extend the original client in a way. That it would fit in nicely as a simulation world for MicroPsi.

A reference to the new class PsiDispatcher has been added to the `minecraftClient`. It's purpose is to check the World adapters data targets frequently and invoke appropriate actions, if necessary. The following figure gives a simplified view of the resulting architecture.

The event loop has been replaced by the function `advanceClient` that is called for every simulation step. For example, one iteration could mean receiving and storing new block data from the Minecraft Server, sending default responses to the received packets, checking the MicroPsi data targets for moving and if necessary send a "Player Position" packet to the Minecraft server.

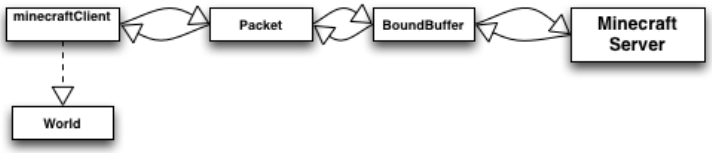

Figure 4.4: *An overview of the most import classes of the `minecraftClient` in respect to the communication with a Minecraft Server*

4.3 Module `minecraftVisualisation`

That being said, the other important part of this project is the visualisation component. It contains classes that provide an interface to the OpenGL context that is the visualisation. This sections gives an overview about what datasources the visualisation uses and how it could be replaced or extended. Inside the world adapter's step function, the visualisation module is called to generate a 3D model of the Minecraft world and the agent within. There are two main reasons for this. 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 effective and pleasurable manor. 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 `minecraftVisualisation` consists of two classes. The class `Window` inherits from pyglet.window.Window and therefore initialises the OpenGL context. For every call of the advanceVisualisation function it updates the 3D model according to world representation inside the minecraftClient and is rendered. A .png snapshot of the framebuffer is generated and displayed in the web interface.

The class Model is also called as a part of the advanceVisualisation function. It contains functions for adding and removing blocks from the OpenGL canvas. The textures

for the blocks are loaded from PNG images and stored as pyglet.graphics.TextureGroup and assigned to vertices when needed.

The visualisation component reads from Spock's internal gameworld representation to generate the 3D model. This means, that from pure Minecraft world data a 3D-visualisation has to be generated from within the MicroPsi Python Code. It should contain a perspective that gives a good overview over the bots environment to forward to the web interface, as well as a the agent's first person perspective, to function as a data source in the future.

The visualisation is in it's core based on "Minecraft" by Michael Fogleman.

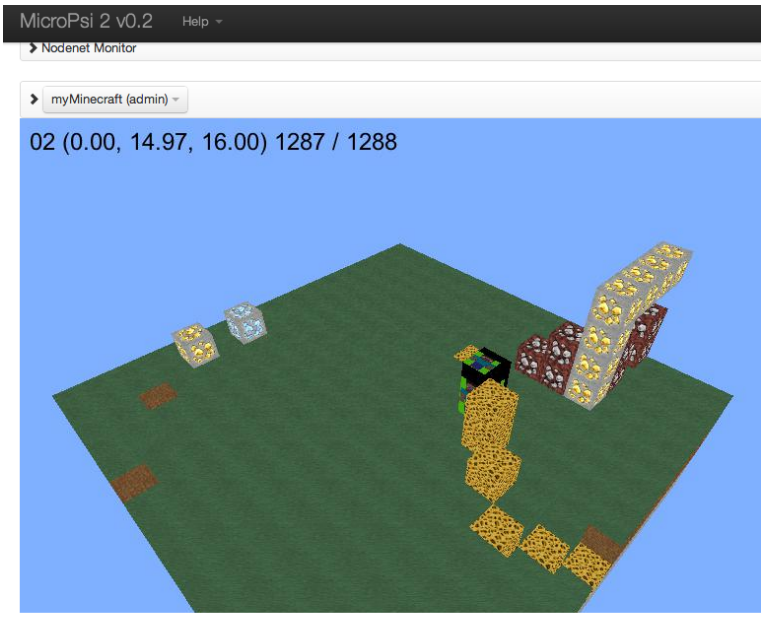

Figure 4.5: *An Image generated by the visualisation component*

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 4.5). Each block gets textures according to it's 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 completely within seconds. The resulting images are exported as JPEG files. Then, they are displayed in the web interface. A refresh rate of six or more images per second creates the impression of a video stream.

Similar to spock, "Minecraft" by Michael Fogleman implements it's own event-loop and -handling. Again, the event loop had to be disassembled and rebuilt as a part of the world adapter's step function, which advances the visualisation with every step.

4.4 Module `minecraftWorld`

The world adapter contains two classes. The class MinecraftWorld inherits from the MicroPsi class world and provides the the assets for the webinterface, an init function and the step function. The class Braitencraft is the name of the actual Worldadapter. In it, the *data sources* and *targets* are defined as dictionaries and an update function advances the life of the client.

Data Targets and sources As a proof of concept, datasources have been defined, that symbolize sensors that detect if a block of diamond in the current section is either before, behind, left or right of the client. The data targets are filled as a part of the world adapter update function as follows:

On the other hand, data targets have been defined for walking moving forwards, backwards, left and right.

4.5 Case Study

As a proof of concept, the Bot is placed in a chunk in which a diamond is placed. The node net has been set up, so that each diamond sensor is connected to the appropriate movement actor, as in figure4.6.

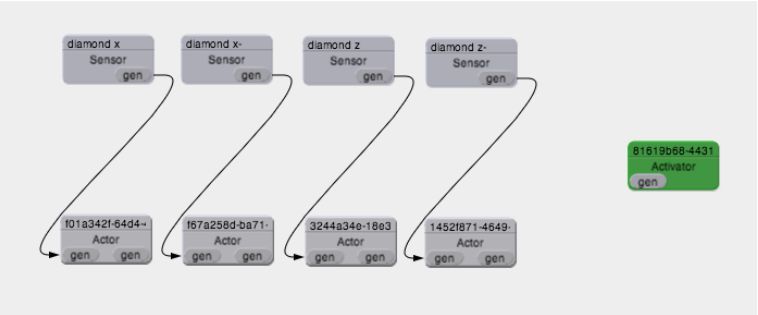

Figure 4.6: *A node net setup for a Diamond-finding experiment*

If we start a simulation like this, the nodes of the sensors that point to the diamond light up green and their activation is forwarded to the actors. The bot move towards the diamond until it it closer than the threshold for the sensors to not detect the diamond anymore — for this experiment the threshold was set to two blocks (see figure 4.7)

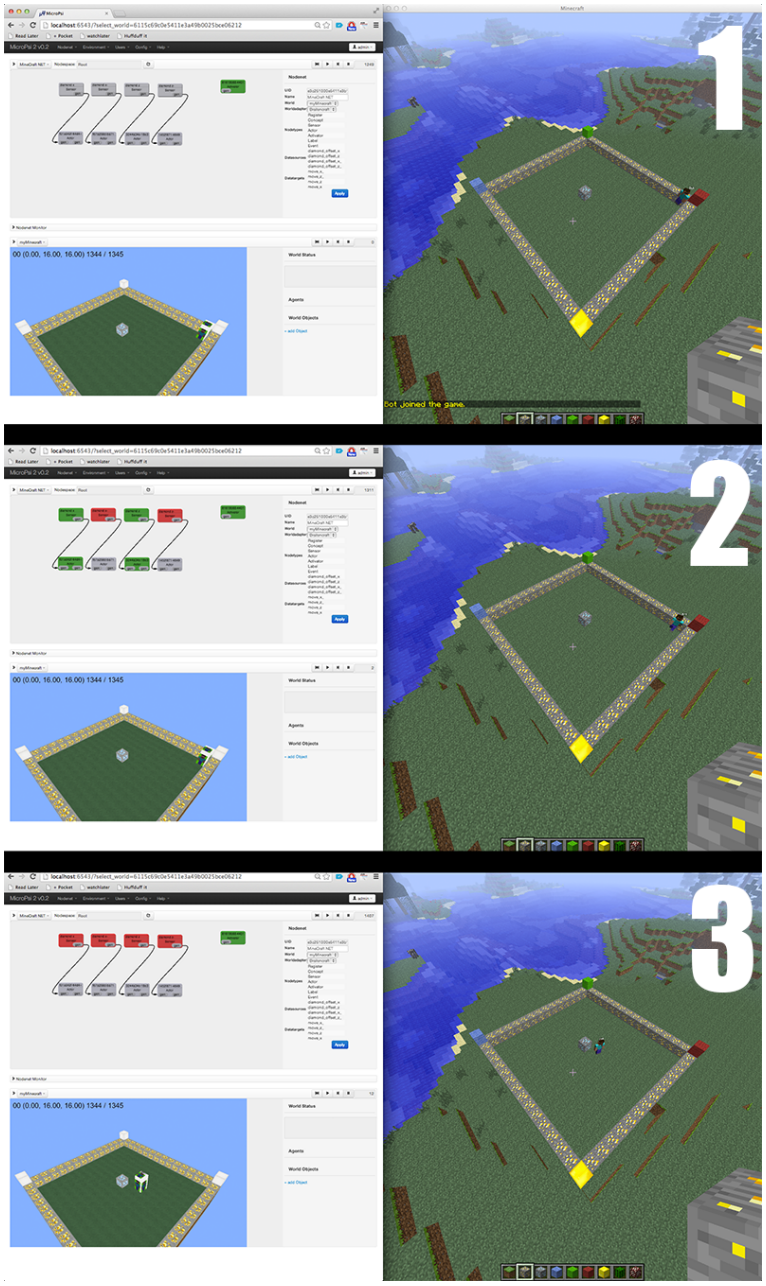

Figure 4.7: *Three screenshots with a Minecraft bot walking towards the diamond*

Chapter 5

Conclusion

After several iterations and trying out different approaches and technologies the Interface is now functional.

The experiment with the simulated Braitenbergvehikel resulted in proofing that Minecraft is usable as a simulation environment.

5.1 What's next

In the future, multiple agents shall interact with the same environment and collaborate with each other. ... what has been learned ... ... what can be done with the new environment ... ... what can be improved? ... ... what other simulation environments could be of interest? ...

List of Figures

List of Tables

Inhalt der beigelegten CD

Die beigelegte CD enthält folgenden Inhalt:

- diese Masterarbeit in PDF Format,
- Videos mit Interview von Fans,
- den Source-Code der Implementierung einer Maschine zur Beantwortung der Fragen aller Fragen. Der Source-Code ist im Ordner *src* zu finden.

Bibliography

[Bac09] Bach, Joscha: *Principles of Synthetic Intelligence PSI: An Architecture of Motivated Cognition*. 1st. New York, NY, USA : Oxford University Press, Inc., 2009. – ISBN 0195370678, 9780195370676

[Bac12] Bach, Joscha: MicroPsi 2: The Next Generation of the MicroPsi Framework. In: Bach, Joscha (Hrsg.) ; Goertzel, Ben (Hrsg.) ; Iklé, Matthew (Hrsg.): *AGI* Bd. 7716, Springer, 2012 (Lecture Notes in Computer Science). – ISBN 978–3–642–35505–9, 11-20

[BB] Balogh, Gergo ; Beszédes, Arpád: CodeMetropolis–a Minecraft based collaboration tool for developers.

[fog13a] http://www.reddit.com/r/programming/comments/1b8a6z/simple_minecraft_clone_in_580_lines_of_python/

[fog13b] https://news.ycombinator.com/item?id=5458986

[git13] https://github.com/superjoe30/mineflayer

[ima13] http://www.minecraftwiki.net/wiki/File:Mob1.png

[mcf13] http://www.minecraftforum.net/topic/1675965-152sspsmp-cubebots-144-now-with-zombi

[mcw13] http://www.minecraftwiki.net/wiki/Chunks

[pro13] http://wiki.vg/Protocol

[pyg13] http://www.pyglet.org/

[sky13] http://skycraft.io/