Statistical Analysis of Player Behavior in Minecraft

Stephan Müller

Semester Thesis
October 2014

Prof. Dr. Markus Gross

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Acknowledgements

I thank my supervisors, Klingler Severin, Mubbasir Kapadia, Seth Frey and Barbara Solenthaler, for their helpful input and continuous support during the project.

I also thank the IT support group from the ITET department for hosting a Minecraft server for us. The student association for informatics students (VIS) and especially Leonhard Helminger helped me organise and promote the Minecraft LAN party.
Abstract

This thesis addresses the development of a framework to collect player data in Minecraft for statistical analysis. We succeeded in creating a complete solution which can be deployed on Minecraft servers and sends the collected data to a centralised server. Using the framework, we collected over 10 hours of player data. That data was used to create heat maps highlighting different player behaviours and a player classifier.
Contents

List of Figures vii
List of Tables ix

1. Introduction 1
   1.1. Motivation ........................................ 1
   1.2. Goals ................................................ 1
   1.3. Related Work ...................................... 2
   1.4. Main Contributions ................................. 2

2. Data Acquisition Options 3
   2.1. Publicly Available Data ............................ 3
   2.2. Recording Data ..................................... 4

3. Framework 5
   3.1. The Epilog Plugin ................................ 5
   3.2. The PrivateWorlds Plugin .......................... 6
   3.3. Ground Truth Collection ........................... 7

4. What to Log 9
   4.1. Bukkit Event Listeners ............................. 9
   4.2. Adding Attributes .................................. 11

5. Getting Data 13
   5.1. Running a Minecraft Server ....................... 13
   5.1.1. DMCA Takedown ................................ 14
Contents

5.2. Recruiting players ......................................................... 14
5.3. Collaboration with Server Admins ........................................ 17

6. Data Analysis 19
  6.1. Ground Truth Collection .............................................. 19
  6.2. Heat Maps ............................................................... 21
    6.2.1. Main World ......................................................... 23
    6.2.2. Custom Maps ....................................................... 23
  6.3. Player Classification ................................................... 25

7. Conclusion 29
  7.1. Future Work ............................................................. 29

A. Communication and Data Storage 31

B. Live Server Stats 35

Bibliography 37
List of Figures

3.1. The framework we used to collect player data ........................................... 6
4.1. Events generated by 30 minutes gameplay .............................................. 10
5.1. Accumulated playtime on the heapcraft server ....................................... 16
6.1. Number of recorded events ................................................................. 20
6.2. Response times and response rate of our ground truth collection ............. 21
6.3. Player position heat map ................................................................. 22
6.4. Block value heat map ................................................................. 22
6.5. Discovery time in map "Prim’s Maze" ................................................. 24
6.6. Deaths in map "Periculum" ............................................................... 24
6.7. Deaths in map "A Light in the Dark" .................................................. 25
6.8. Classification matrix ................................................................. 26
A.1. Raw data of logged events .............................................................. 32
B.1. Live server stats section of the companion website ............................. 36
List of Tables

6.1. Confusion matrix validation set .......................................................... 27
6.2. Confusion matrix training set .............................................................. 27
List of Tables
Introduction

1.1. Motivation

Computer games have a huge potential for scientific research. Extensive datasets can be created by recording a player’s every move and action. That data represents behaviour of actual humans in a controlled environment, making it especially valuable to social sciences.

Minecraft is very well suited for player analysis. With well over 15 million sold copies\(^1\), it has a large user base and a very active community. In its basic form, the game is an open world sandbox with no obvious goals. The nature of the game motivates players to explore, mine for resources and build infrastructure. As soon as multiple people start playing on the same server, communities and even economies start emerging. The game can be modified with custom code, allowing to introduce new game mechanisms and put players in specific situations. It is possible to build new levels or any kind of virtual world, either in game or programatically. The large variety of online servers with different policies and modifications could very well turn out to be sociological goldmines if enough player data is made available to researchers.

1.2. Goals

The goal of this project was to investigate the possibilities to analyse player behaviour Minecraft. After exploring different ways to acquire data, we developed a Minecraft plugin which is able to record the actions of players on a game server and send the data to a centralised database for storage. A Minecraft server had been set up in order to collect data and test the plugin. Ad-

\(^{1}\)https://minecraft.net/stats.jsp
1. Introduction

Additional plugins were developed and deployed, allowing us to collect ground truths and record data of single player behaviour in custom maps. The resulting data was then used to detect different classes of player behaviour and create some interesting visuals.

This thesis should provide an overview and possible entry point for people interested in doing player analysis in Minecraft. For that reason we not only built our tools to be universal and reusable, but also included some of our experience collecting the data to analyse.

1.3. Related Work

Nicolas Ducheneaut and Nick Yee [DY13] did similar work collecting player data in World of Warcraft. They used client plugins to log data about other players. In addition, they used data provided by the game’s publisher (Blizzard Community API) and survey data from players.

Castronova et al. [CRK13] made a case for using games for social science experiments. In contrast to methods like historical analysis or laboratory experiments, experiments conducted within computer games are inexpensive to scale and enable tinkering. They also used Blizzard’s public World of Warcraft data for their case study. Even though it is not mentioned directly, they make very good points for using Minecraft in social science.

William Sims Bainbridge [Bai07] highlights the benefits of setting up laboratory experiments in virtual worlds. Virtual labs make it easy to reach large numbers of people across sociocultural boundaries and do experiments lasting a long time. The game Second Life was used to build the experiments.

The most player data analysis is probably still done by game developers themselves with the main goal of improving the player experience. Their results unfortunately are usually kept secret to ensure an advantage over the competition. While the data is traditionally collected from selected playtesters in special playtesting labs, the recent Destiny Public Beta² has dramatically increased the amount of collected data by basically allowing everyone to become a playtester. Player data in online games often keeps being analysed even after their official release, but changing finished games is risky and has to be done very carefully [CRK13].

1.4. Main Contributions

This thesis makes the following main contributions:

1. A data collection framework for capturing Minecraft player behavior which can be seamlessly deployed on existing or new servers
2. Visualization tools to help glean meaningful information from behavior logs which can be used to inform server administrators and players alike
3. Ground truth data collection and preliminary analysis for classifying players based on their behavior

²http://destiny.wikia.com/wiki/Destiny_Public_Beta
Data Acquisition Options

The easiest way to get player data is usually by being or collaborating with the game developer. In cases where this is not possible there are still several ways to access information about players. We put together an overview for some of those options in Minecraft.

2.1. Publicly Available Data

Mojang, the developer of Minecraft, started collecting\(^1\) user data in 2012. Some of it got published\(^2\). The data however is mostly\(^3\) about hardware information and performance. There is no data collected about actual gameplay.

Minecraft also collects some player related statistics\(^4\) automatically. The data consists mostly of simple counts like total number of deaths or distance walked while sneaking. Even though the collected variables seem mostly anecdotal and can’t be separated by time, it could be used for player classification.

Minecraft servers implement the Source RCON Protocol\(^5\) which is used by most server listing websites to display things like the current and maximum allowed number of online players. There is open source code\(^6\) to simplify such queries. A script could be built to query several server and store the data for analysis over time. The data which can be acquired by this method

\(^1\)http://notch.tumblr.com/post/16463856106/a-poll-on-letting-us-snoop
\(^2\)http://stats.minecraft.net/
\(^3\)http://wiki.vg/Snoop
\(^4\)http://minecraft.gamepedia.com/Statistics
\(^6\)https://github.com/xPaw/PHP-Minecraft-Query
2. Data Acquisition Options

is however very limited.

2.2. Recording Data

There are a lot of existing server plugins that log player data. A very prominent category of those plugins are anti-griefing tools. Some of them try to record all user actions affecting the world state, so they can perform rollbacks based on individual users. A prominent example is Prism\(^7\) which is open source\(^8\). Many more data logging plugins can be found on plugins.bukkit.org, but a lot of them are not being actively developed and the developers tend to be unresponsive, e.g. when requesting source code. Judging by the popularity of those plugins, there must be an enormous amount of collected user data out there.

Player data could also be recorded by modding the clients of individual players. Installing client mods however is not trivial\(^9\) and requires some technical skills and willingness to invest time by the players. This method has the benefit of being able to log a player across several servers. In contrast, server side logging will lead a bigger number of different players and data that is easier to compare since players are acting in the same environment.

A modified client could also be used to record other players on the same server (like done by Nicolas Ducheneaut and Nick Yee [DY13]). World changes and many player actions are broadcasted to other clients in order to accurately display the world and animate other players. A dissection of communication protocol between server and client is available online\(^10\). However, the clients only receive information about nearby events and covering a theoretically infinitely large world might be difficult. But covering certain hotspots on public servers should be possible using this method.

---

\(^7\)http://dev.bukkit.org/bukkit-plugins/prism/
\(^8\)https://github.com/prism/Prism
\(^9\)http://minemum.com/installing-mods
\(^10\)http://wiki.vg/Protocol
Framework

Fig. 3.1 shows the different components in our framework and how they interact. The Epilog plugin can be installed on existing servers and always sends the collected data to our logging server. The two other plugins use the Epilog plugin to send data to our database.

3.1. The Epilog Plugin

After evaluating the different ways of gathering player data, server-side plugins seemed to be the way to go. Not only do they allow us to log exactly what we want, they also allow us to capture whole servers at once, allowing us to collect a lot of player data quickly. We decided to develop Epilog, our own server plugin. Compared to existing plugins, ours has the following advantages:

- automatically send data to a centralised server
- log additional data other plugins do not, e.g. player position
- lightweight, does not add user commands, no interference with existing plugins

The plugin can be used to log data on servers we run ourselves as well as on existing servers. All we have to do is convince the admins to install it. Once active, it logs all players on the server without additional actions required on their side.
While there is no official Minecraft API for writing mods or plugins yet, there are couple of projects to fill the gap. We used Bukkit to write our plugin. They decompile the original Minecraft server binary and add their own code to allow loading plugins written against their own plugin api. While the plugins can’t be used on a vanilla Minecraft server, many servers use Bukkit compatible software.

3.2. The PrivateWorlds Plugin

The Epilog plugin does a good job at collecting player data from servers where typically multiple players play together in one single world. But sometimes it is desirable to record players playing alone. If a player gets provided their own world within a server, their behaviour will likely be just as if they are playing the game offline in single player mode. There are also thousands of handcrafted Minecraft maps online which are are supposed to be played alone. Recording player behaviour on those maps can give useful feedback to the map designers. Social experiments can be implemented as custom maps, providing easy access to millions of minecraft users.

We developed a plugin named PrivateWorlds to automatise the process. It provides one single command (/pw) the players can use to be teleported into a hub where they can choose from a variety of maps to play in. The hub is a dynamically generated room inside the game. Upon pushing the button associated with a map, a new copy of the map gets loaded into a new virtual world and the player gets teleported into it. If the player has already played that map she can

---

1 http://minecraft.gamepedia.com/Plugin_API
2 http://minecraft.gamepedia.com/Mods
3 http://bukkit.org/
4 https://minecraft.net/stats
choose to resume where she left off instead of creating a new instance. The `/pw` command can be used to leave a private world at any time. Every time a player leaves a private world they have to rate the map on a scale from 1 to 4 (bad, neutral, good, excellent).

The gameplay within the main server world and the different private worlds shouldn’t affect each other. For that reason the player states (including amongst others inventory, experience points, spawn location, velocity, health or being on fire) are saved when leaving a world and restored upon reentering. When entering a world for the first time the player’s state is set to an initial state (e.g. empty inventory, not burning). Having different player states in different worlds prevents things like players bringing things from one world to another or teleporting to another world shortly before dying to save themselves.

Pushing a button in-game to trigger an action seemed more user friendly compared to typing commands to the console. But including such buttons within the private worlds would require modifying the maps. A console command also allows to leave a world at any time. We didn’t provide a button to enter the hub on purpose to make sure the players in the private worlds are able use the console and know how to leave the world again. The usage of the PrivateWorlds plugin was explained on the companion homepage and on signs inside the hub and at the spawn of the server’s main world.

### 3.3. Ground Truth Collection

In order to create and verify our player categorization we created a plugin to randomly ask people what they were doing via in-game chat. The sampling method was inspired by the work of Csikszentmihalyi [CL87]. The reminder message “$PLAYERNAME, what are you doing? type `/do help`” was sent to players every 3 to 13 minutes. The “/do help” command provides more information about how to use the command. A more detailed description can be found on the companion website. Players can also use the command to tell what they are currently doing without receiving a reminder or even turn them off should they feel distracted.

The options are explore, mine, build, fight and other. Players can select them by using the first letter, e.g. “/do b” for building. Our options are roughly based on the Bartle Test\(^6\). We however tried to use terms that are unambiguous and easy for the players to understand. Socialising for example, while one of the defining factors of the Bartle Quotient, could be interfering with other options. Two players could for example be fighting monsters together. Also, socialising can be detected by other means, for example by player proximity or chat activity.

---

\(^5\)For a more normal-like distribution, three random numbers are averaged.  
\(^6\)http://en.wikipedia.org/wiki/Bartle_Test
3. Framework
What to Log

As part of building the player classification we had to figure out what kind of data to log with our Epilog plugin. We tried to capture as much player data as possible. This will also allow us to use the plugin and the collected data for other projects in the future. However, to keep the size of the collected data reasonable, we had to choose what kind of data to include.

4.1. Bukkit Event Listeners

Bukkit plugins can register event listeners. Since we are primarily interested in player data, we decided to start by logging events that can be associated with a player. In order to get a list of those events, we used a script to collect all classes inheriting from the `Event` class inside the Bukkit API source code¹. Then we chose all events having a `getPlayer()` method.

Unfortunately, not all events which can be associated with a player have a `getPlayer()` method. Some events can also be triggered by non-player entities. We therefore hand picked some additional events which can be triggered by players. A more elegant solution would be to select all event classes having a method which returns an object of type `Player` or a superclass thereof. Our solution however required far less code.

We ended up with a list of 75 events. The next challenge was to figure out which of those events are actually useful and if more data needs to be stored for individual events in addition to the event name. In order to know where to begin, we recorded about 30 minutes of intensive gameplay and started analysing the most frequent events.

As expected, `PlayerMoveEvent` dominates the statistic. Move events are generated every

¹https://github.com/Bukkit/Bukkit
4. What to Log

Figure 4.1.: Events generated by 30 minutes gameplay

- PlayerMoveEvent: 11379
- PlayerAnimationEvent: 6460
- PlayerStatisticIncrementEvent: 1279
- PlayerInteractEvent: 744
- BlockDamageEvent: 532
- BlockBreakEvent: 340
- PlayerPickupItemEvent: 326
- PlayerToggleSprintEvent: 170
- PlayerItemHeldEvent: 164
- PlayerExpChangeEvent: 70
- BlockPlaceEvent: 68
- InventoryCloseEvent: 58
- InventoryOpenEvent: 44
- FurnaceExtractEvent: 38
- FoodLevelChangeEvent: 33
- EntityRegainHealthEvent: 23
- PlayerToggleSneakEvent: 20
- EntityDamageByEntityEvent: 20
- PlayerVelocityChangeEvent: 14
- PlayerTeleportEvent: 10
- PlayerLevelChangeEvent: 7
- EntityDamageEvent: 4
- PlayerItemConsumeEvent: 4
- PlayerAchievementAwardedEvent: 2
4.2. Adding Attributes

game tick if the player position has changed, which is 20 times per second while walking. This might be one of the reasons why most plugins don’t log player movement. We considered reducing the amount of log data by only saving move events after a time- and/or distance threshold or compressing the data using the Ramer–Douglas–Peucker algorithm\(^2\). But since the amount of data per event is relatively small (5 floats: x, y, z, yaw, pitch) we kept logging each movement event individually.

The second most frequent event represents a player swinging their arm. This animation is almost always followed by the event causing the arm swing, e.g. \texttt{PlayerInteractEvent} or \texttt{BlockDamageEvent}. Since the event was rather frequent and contained almost no additional information, we decided to not log it. Same goes for the \texttt{PlayerStatisticIncrement} Event and some others later on.

4.2. Adding Attributes

Some events get a lot more useful if they include additional information. For example when logging a \texttt{PlayerToggleSneakEvent} it would be nice to know whether sneaking was turned on or off. Many events can be associated with a block and knowing its position and type makes sense intuitively. This information gets added automatically if the event class has a method called \texttt{getBlock()}.

We continued logging gameplay and trying to trigger new events. For some of them we decided to add additional information manually. A couple of events seemed to be triggered too frequently, for example \texttt{EntityDamageByEntityEvent} was often triggered 4 times at once while subtracting the damage from the health only once. When picking up items, the amount always returned 0 and \texttt{InventoryOpenEvent} was only triggered in certain cases. We reported some of the bugs to the Bukkit project and fixed a couple ourselves. There might also be some additional bugs in our own code. In the end, we accept that the recorded data has some errors and will take it into consideration when analysing the data. Time will tell whether additional data is required for certain events.

In addition to the native Bukkit events some new events have been created. For example a \texttt{PlayerItemInHandEvent} gets triggered every time the item a player changes the item in hand, containing the name of the new item as meta data. The item in hand can potentially change after native events like \texttt{PlayerPickupItemEvent} or \texttt{InventoryCloseEvent}. \texttt{EntityDamageByEntityEvent} is broken down into four new events, depending on whether any of the entities represents a player.

More information about the plugin and examples of raw event data can be found in Appendix A.

\(^2\)http://en.wikipedia.org/wiki/Ramer-Douglas-Peucker_algorithm
4. What to Log
Getting Data

Collecting enough data for statistical evaluation turned out to be harder than assumed. To be on the safe side, we decided to pursue two strategies at once: running our own server and asking other server admins to install our logging plugin. In the end we succeeded in recruiting enough players for our server in order to do a variety of statistical analysis.

5.1. Running a Minecraft Server

Running a server ourselves has a lot of advantages. For example, the plugin can be installed from the beginning, generating a complete log over the whole history of the server. We can do small experiments, test new features and are able to quickly deploy fixes in case something goes wrong.

We set up a new Minecraft server named “heapcraft” featuring a randomly generated world. The server difficulty was set to easy and we disabled the ability of mobs to modify the world (e.g. exploding Creepers would not create holes). Other than that, the server is using the default configuration. We hoped this approach would drive the players to do all kinds of different activities and not push them into a particular direction.

The Epilog plugin has been installed from the beginning and has logged the compete history of the server, with the exception being down for about two hours due to user error after deploying a new version. The PrivateWorlds plugin has also been installed, featuring three custom adventure maps. A new plugin to collect ground truth data over in-game chat was added after a couple of weeks (see section 6.1).
5. Getting Data

5.1.1. DMCA Takedown

All our plugins are written against the Bukkit plugin API which is currently affected by copy-right issues. The Bukkit project received a DMCA takedown request\(^1\) regarding CraftBukkit September 3rd 2014. CraftBukkit is a modified minecraft server version required to run our plugins. The takedown affects many servers using either CraftBukkit directly or forked projects like Spigot\(^2\). The reason for the takedown is CraftBukkit being distributed containing closed source code from the original Minecraft server binary, while also containing code licensed under the GPL which forbids major components that are closed. Existing servers using CraftBukkit do not seem to be affected by the takedown directly, but will probably not get any future updates.

Sponge\(^3\) is an alternative Minecraft API that is not affected by the DMCA takedown. Pore\(^4\) is a Sponge plugin currently being developed with the goal of making Sponge compatible with the Bukkit API. Mojang could also soon be releasing their own official Minecraft API which could be Bukkit compatible.

5.2. Recruiting players

One drawback of running a server is having to recruit players. Supply of Minecraft servers has grown tremendously since its initial release and simply having a server does not attract players anymore. Server listing sites promote addresses for 100’000s of different servers.\(^5\) While trying to bring new people to our server we noticed a certain server loyalty. Players spend a lot of time building infrastructure and communities. So checking out a new server becomes an investment and needs a matching incentive.

Ideally, the incentive to play on our server would be having an extraordinary amount of fun. Considering the open nature of the game, there are a lot of options to reach that goal. One way is having an active community, building interesting things. So the problem can be reduced to creating a good initial community and thereby good social norms. According to Castronova\(^[CRK13]\), this should attract new players and synchronize their behavior. The following paragraphs will summarise the way we reached that goal in hope of helping researchers with similar endeavors.

The primary tagline for our server was “play Minecraft for science”. We built a website containing some information about the project and our Minecraft server. The website features some live statistics fetched from our recorded data like the total play time or the number of diamonds mined. More information about the live stats are available in section \(\text{B}\). It also includes a live heat map of player positions since launch. More information about the heat map can be found in section 6.2.1 and Fig. 6.3. When choosing the information to display we tried not affecting the gameplay too much. The heatmap for example only showed a limited area around the spawn

\(^1\)http://dl.bukkit.org/dmca/notification.txt
\(^2\)http://www.spigotmc.org/
\(^3\)http://spongepowered.org/
\(^4\)https://github.com/LapisBlue/Pore
\(^5\)minecraftservers.org tracks almost 40’000 servers and minecraft-server-list.com cites to have monitored over 250’000 servers. However, many servers are not currently online.
5.2. Recruiting players

point, so people would still be able to build secret houses further away.

We told people mostly about the website instead of revealing the server address directly. That way we had a more convenient way of communicating with our users and giving them information about our study and the server.

Reaching out to friends brought some players to the server but failed to have any sustainable effect. Posts about the server on a popular Minecraft forum\(^6\) got almost completely ignored. (The majority of active threads were about existing communities looking for new members. The high amount of new posts pushes those without responses into oblivion after a couple of days.) Advertising servers on other servers (e.g. per in-game chat) is frowned upon and considered spam which can lead to being banned. The same goes for Minecraft centric IRC channels like `#Minecraft` on `irc.esper.net`. They tend to offer specialised channels which are crowded similarly to the previously mentioned forum. However, I had some success finding interested players on IRC channels not directly related to Minecraft and on more obscure forums.

Even though we were able to collect a couple of hours of gameplay, the server did not gain any real traction. We continued our efforts with two additional ventures: offering chocolate to lab members and their kids and friends for hours spent playing on the server and organising a Minecraft LAN party. The LAN party was organised in collaboration with a student association and announced over a mailing list reaching about 700 informatics students.

As can be seen on Fig. 5.1, the server starts taking off after sending the announcement mail on August 16th. It turns out that we didn’t only attract players interested in the LAN party but several Minecraft players excited to play with other people from the university or just kind enough to help our project. For the actual LAN party, nine people showed up. Most of them haven’t been playing on the server before. Nevertheless, the party was able to add a solid two days of recorded gameplay. One day later, we reached the total of ten days of playtime which we had previously set as a threshold for starting the statistical analysis. At that time, 46 different players had visited our server.

A couple of days later, people seemed to lose interest in the server again. Since we had already reached our data collection goals we did not further investigate potential reasons. Not upgrading the server to the newest Minecraft versions or people going on vacation before the beginning of the new semester could be one of them.

Additionally, 21 chocolate bars have been claimed for 21 hours of gameplay. The mail announcing our chocolate campaign went out August 15th, one day before the LAN party announcement. Since we didn’t track individual users we were unable to separate the playtime by incentive. But it should be safe to assume that telling a large number of people about a Minecraft server they can somehow relate to (e.g. by being supported by their student association or benefiting a project within their department) will attract players. Even if their interest might only be temporary.

\(^6\)http://www.minecraftforum.net/
Figure 5.1: Accumulated playtime on the heapcraft server
5.3. Collaboration with Server Admins

Installing the plugin on existing servers also has its challenges. Especially admins of large servers are concerned about performance impact and stability issues. Downtime and unhappy players can hurt the servers image and eventually even lose them money. Small servers on the other hand tend to have little player activity and require a long logging period to get good data.

We started looking for admins to install our plugin the same way we started looking for new players for our server. First we wrote some form posts without much success. After our server started to gain traction, data from other servers became less important. We still had the information about the plugin on our companion website and got a couple of emails from admins willing to participate.

When we started evaluating our collected data, we had only 4.6 hours worth of gameplay from other servers. This was not enough data to investigate differences between servers (see section 7.1). Considering differences between those servers, combining the data with the logs from our own server might still have affected the results. We therefore decided to ignore other servers for now.

The participating admins volunteered because they wanted to help our project. In order to get more admins to include their servers, we may have to provide additional value. Offering heat maps like we used on our companion website and other interesting information about their server derived from the collected data could be a good incentive. Also making the source code available could provide some ease of mind especially to admins of larger servers.
5. Getting Data
Data Analysis

We started collecting data on a new server with a randomly generated world July 13, 2014. 42 days later (August 24, 2014) we had recorded 10 days worth of gameplay and took a snapshot of the database to base our initial analysis on. 30 players created more than 1000 events, which equals about 1 hour of gameplay. Players with less activity are ignored.

Fig. 6.1 shows the number of events included in our dataset. Not shown are 926’7043 Player-Move Events. We also excluded events occurring less than 100 times and events with a very strong correlation to another event (e.g. InventoryOpenEvent and InventoryCloseEvent). Furthermore some events we didn’t think would add much additional information for our purpose were ignored (e.g. PlayerLoginEvent).

6.1. Ground Truth Collection

The plugin to record ground truth was installed and started collecting data August 18th, two days after the LAN party announcement mail was sent. At the time of the snapshot, 1477 reminders had been sent and 529 self-classifications received. The distribution was 202, 93, 28, 132 and 74 classifications for build, explore, fight, mine and other. The number for “fight” is notably low. This doesn’t necessarily mean that our server is especially peaceful. Fights often don’t last as long as actions from the other categories and players might be too busy to use the console during a fight. They might already have returned to their previous task when answering or even noticing the reminder.

After looking at how long it takes player to send a classification after receiving the reminder (see Fig. 6.2), we decided to consider them a response if the time difference is smaller than 60 seconds. This led to an average response rate of 0.314. The histogram in Fig. 6.2 on the right
6. Data Analysis

Figure 6.1.: Number of recorded events
Figure 6.2: Response times and response rate of our ground truth collection

shows that five players ignored all the requests, but the bigger part of players responded to at least some of them. Furthermore, one player deactivated the reminders temporarily and two players permanently. 65 classifications couldn’t be associated with a request. Their distribution in respect of time seemed random enough to still include them in our ground truth data set.

6.2. Heat Maps

Heat maps are an excellent tool for visualising spatial information. In Minecraft, knowing where things happen can lead to deep insights about the behaviour of players and the nature of the map. They help to recognise patterns and locate interesting points. In this subchapter we do not try to answer specific questions. Instead we demonstrate how heat maps can be used for descriptive player analysis.

We scaled the data matrices with $\text{sgn}(M) \cdot \log(\text{abs}(M) \cdot a + 1)$ to make the resulting images more readable. $M$ represents the two dimensional data matrix, $a$ is scaling factor used to enhance image contrast, similar to gamma correction. The function maps real numbers to a scale between -1 and 1. The result is similar to using a logarithmic scale, but works for both negative and positive values.

Every pixel represents the area of one block.
6. Data Analysis

Figure 6.3.: Player position heat map

Figure 6.4.: Block value heat map
6.2. Heat Maps

6.2.1. Main World

Fig. 6.3 shows where players spent their time on the server. Darker colors mean more time. We used a logarithmic scale so we would not only see areas with high activity, but also paths walked by individuals. Only a limited area around the spawn point is shown because including the whole active area would obscure details. We ignored players idle for more than 1 second to avoid getting hot spots when players leave the keyboard to wait for daytime or their plants to grow.

Fig. 6.4 shows the value of blocks placed (dark) and removed (bright) by the players. Buildings can be recognised as dark rectangles. They get darker by either being tall or being made of expensive materials. Mines and farms leave bright traces from removing stone, dirt and other resources or harvesting plants. Paths of dark spots indicate torches placed while exploring caves.

The block value was taken from blocksandgold\(^1\). They use a trading system based on a virtual currency to determine the value of items. The price list gets updated daily. We took the values from October 6th 2014. Players on our server are likely to value blocks differently, but taking data from a different, well working economy is expected to be a good approximation.

Because the price list sometimes differentiates between using information we did not log, some additional mapping is required. Anvils, for example, can have different levels of damage, decreasing its worth. In such cases we took the lowest value, avoiding the risk of assigning the value of a rare variation to an otherwise cheap block. Furthermore, the map only includes blocks that are placed or mined. Blocks or other items stored in chests are ignored.

6.2.2. Custom Maps

The following heat maps were created with data generated by users playing on custom maps using the PrivateWorlds plugin.

Fig. 6.5 shows a map we created ourselves using a random maze generator\(^2\). The map has been played by 6 different people. We calculated the time at which any of the players first reached a position after starting playing the map. The result visualises how the map got explored. The map starts at the top left corner where the paths are bright indicating them being discovered first. On the bottom right corner there is a garden representing the end of the game which only got discovered after long exploring. Sometimes the tips of dead ends show a significantly darker color. This indicates them being entered again for closer inspection.

The next two maps show player positions, overlaid with positions where the players have died (red). The first map in Fig. 6.6 is created by recording player data from the map Periculum\(^3\). In Fig. 6.7 A Light in the Dark\(^4\) was used. Both maps had been played by 8 different people.

Difficult parts on the map can be identified by dark colors (players spending a lot of time at

\(^1\)http://www.blocksandgold.com/en/minecraft-item-id-price-list/
\(^2\)http://en.wikipedia.org/wiki/Maze_generation_algorithm Randomized Prim’s algorithm
\(^3\)http://www.minecraftmaps.com/adventure-maps/periculum
\(^4\)http://www.minecraftmaps.com/parkour-maps/a-light-in-the-dark
6. Data Analysis

**Figure 6.5.:** Discovery time in map "Prim's Maze"

**Figure 6.6.:** Deaths in map "Periculum"
6.3. Player Classification

Using the ground truth data, we built a classifier to predict what a player is doing based on our logged data. We ended up with 29 events by removing events that did not seem to matter or did not occur frequently enough to add useful information. `PlayerMoveEvent` is transformed into move distance and the toggle events for sneak and sprint is transformed to time spent sneaking and moving respectively. The other events are represented by simply counting their occurrence.

The classification happens with accumulated data over three minutes. Based on the time it took players to answer the classification reminders, the time seemed appropriate. Accumulating data over a longer period has the advantage of capturing a more average view of the behaviour, even with fewer samples. Shorter periods on the other hand reduce the risk of capturing several different behaviours at once. Specific tasks usually take well over three minutes. Not knowing whether the players would state what they were doing up to now or if they just started doing something new, we accumulate the events 1.5 minutes before receiving their declaration until 1.5 minutes after.

Only data from our own server was used, excluding gameplay inside custom maps. We created two data sets with the method mentioned above: a training set containing all 298 entries at the

---

**Figure 6.7.: Deaths in map "A Light in the Dark"**

the same spot) and by red pixels (players have died). This data can be used to identify areas where the map is confusing or too difficult. Heat maps of this kind can be used to distribute the difficulty of a custom map more evenly so a player stays challenged most of the time without getting frustrated by overly difficult parts.
time of the snapshot, and a validation set with 103 entries after the snapshot was taken. A third set was used for normalisation, containing the complete data of the snapshot. The beginning of the three minute intervals was set to the time of the first event after the previous period had ended. This also resulted in ignoring periods where a player was completely idle or logged out.

The data was normalised based on our normalisation set. Every variable was shifted and then scaled to have zero mean and unit variance. This allows us to compare the different variables to each other and base our math on deviation from average gameplay. The same shifting and scaling was then applied to the two other data sets.

A classification matrix is calculated by first averaging all entries for a certain action in the training set. The resulting row vectors describing an average action are then scaled to have unit length, producing the matrix shown in Fig. 6.8. Casual inspection shows that the results make sense intuitively. Variables one would associate with a certain action stand out reasonably well. Signal theory predicts that random noise will average to zero. Unfortunately, this only works if the noise is statistically independent of the signal. Fighting for example seems to occur disproportionately often during exploring, resulting in a clearly noticeable fighting component inside the exploring vector. Trying to minimise manual intervention to keep the approach universal, we did not try to separate the variables any further.

The classifier works by comparing the direction of the vector representing a datapoint to the vectors in the classification matrix by taking their scalar product. Since both vectors are first normalised to unit length, the result will be between -1 and 1, indicating how much the vectors point to the same direction. Multiplying the data set with the classifier matrix will result in a new data set, giving a rating for every class to each datapoint. The class with the highest value

---

5 We realise that separating the training and validation set by time might introduce systematic errors. In order to test the classifier, the samples should have been assigned randomly.
Table 6.1: Confusion matrix validation set

<table>
<thead>
<tr>
<th>true</th>
<th>build</th>
<th>explore</th>
<th>fight</th>
<th>mine</th>
</tr>
</thead>
<tbody>
<tr>
<td>build</td>
<td>0.367</td>
<td>0.107</td>
<td>0.013</td>
<td>0.047</td>
</tr>
<tr>
<td>explore</td>
<td>0.013</td>
<td>0.093</td>
<td>0.007</td>
<td>0.007</td>
</tr>
<tr>
<td>fight</td>
<td>0.000</td>
<td>0.027</td>
<td>0.020</td>
<td>0.000</td>
</tr>
<tr>
<td>mine</td>
<td>0.047</td>
<td>0.047</td>
<td>0.000</td>
<td>0.207</td>
</tr>
</tbody>
</table>

Table 6.2: Confusion matrix training set

<table>
<thead>
<tr>
<th>true</th>
<th>build</th>
<th>explore</th>
<th>fight</th>
<th>mine</th>
</tr>
</thead>
<tbody>
<tr>
<td>build</td>
<td>0.307</td>
<td>0.038</td>
<td>0.029</td>
<td>0.076</td>
</tr>
<tr>
<td>explore</td>
<td>0.033</td>
<td>0.107</td>
<td>0.029</td>
<td>0.027</td>
</tr>
<tr>
<td>fight</td>
<td>0.004</td>
<td>0.009</td>
<td>0.049</td>
<td>0.000</td>
</tr>
<tr>
<td>mine</td>
<td>0.044</td>
<td>0.029</td>
<td>0.020</td>
<td>0.200</td>
</tr>
</tbody>
</table>
6. Data Analysis

is then selected as output.

Another way of looking at the classifier is as correlation receiver. The rows of the classification matrix represent matched filters. Since this is a well known and widely used linear system, its behaviour is very well known and the results are easy to interpret. Actual gameplay usually features combinations of the different actions. If we leave out the last step selecting the maximum, we get ratios for each filter individually. In some cases this information might be more valuable. Not having competing classifiers also offers greater freedom when adding additional filters.

Results of the classification are shown in Table 6.1 and Table 6.2 (the numbers are divided by the total number of classifications). The classifier had a success quotient of $\frac{103}{150} = 0.69$ when applied to the validation set and $\frac{298}{450} = 0.66$ applied to the training set. Having that much information about the players we expected the numbers to be higher. But the relatively low numbers do not necessarily point to bad quality of the classifier. A player declaring explore could have been mining most of the measured 3 minutes. Or building a house inside a mountain could (and probably should) be measured as mining. Before trying to improve the classifier based on the available data we recommend further analysing the gameplay leading to misclassifications, e.g. by watching some players play. It should also be noted that a random classification would result in a success quotient of 0.25. Our classification is more than 2.5 times better.

Standing out with a large number of false negatives is once again fighting. On the validation set, more fight data points got classified as explore than fight. While this could just be an anomaly caused by the small number of cases, previously mentioned warning signs may suggest there being an actual problem. The averaged data used for the classification matrix had clear signs of fighting mixed in the exploring row and the usually short duration of fighting compared with the other actions means there often might be other actions included in a data point declared as being fighting.
Conclusion

We successfully built a framework for recording gameplay on Minecraft servers and were able to record over 14 days worth of player data. The collected data provided interesting insights into player’s behaviour. By creating heat maps we were able to recognise different kinds of player activity. A classifier allowed us to distinguish between farming, fighting, mining and exploring with about 65% accuracy.

7.1. Future Work

The Epilog plugin is likely to be of value to other researchers. Some additions like the ability to track items players are keeping in their inventory or placing in chests or keeping track of arrow ownership to analyse fights in more detail would certainly improve that value. In light of the uncertain future of the Bukkit project, a rewrite to a new API could be considered.

The method we used to classify players over time could be extended to distinguish between players displaying different behaviour in general. Combined with more data from different servers, correlations between server policies, plugins, communities, player types and player satisfaction are very likely.

Using Minecraft as a laboratory, a lot of questions related to social science can be investigated. Analysing virtual economies, ownership, governance or civilisation, to name a few examples.

The PrivateWorlds plugin lets players rate maps after playing them. This information combined with all the recorded gameplay can be used to analyse different maps. Knowing what players like about maps, an algorithm could be developed rating maps automatically.
7. Conclusion
Communication and Data Storage

The EpiLog plugin captures a lot of data that has to be collected eventually. While simply walking around generates 20 events per second, a user can easily generate over 50 e.g. by farming which can include walking, breaking and picking up wheat at the same time. After collecting 10 days worth of data the average amount of events per active user was around 12 per second.

We were looking for a robust and easy to build solution for sending the collected data to a logging server. Experience has shown that maintaining a persistent connection comes with lots of engineering challenges. Hence we decided to send the data as HTTP POST requests. This allowed us to save the data on a regular web server and conveniently write the logger code in PHP.

Instead of using a binary protocol like Google’s Protocol Buffers\(^1\) we send our data as JSON for greater flexibility. In order to reduce the overhead of sending HTTP request the plugin sends the events in batches every 10 seconds. Sending several events together also improves compression rates, especially for the rather verbose JSON protocol we are using. In the end, the amount of data sent to the server was comparable with a specialised binary protocol.

Fig. A.1 shows an example of 5 events as they are sent to the server. The following variables common to all events were omitted:

- **token** identifies the server where the event occurred, represented by IP address and port number
- **eventClass** (generic, damage, move, server) tells the server which table to use for storing the event

\(^1\)https://developers.google.com/protocol-buffers/
A. Communication and Data Storage

Figure A.1: Raw data of logged events

```json
{
    "event": "BlockBreakEvent",
    "material": "STONE",
    "blockX": 444,
    "blockY": 12,
    "blockZ": 908
},
{
    "event": "PlayerMoveEvent",
    "x": 443.3000000119208,
    "y": 11,
    "z": 908.4286558138695,
    "pitch": 37.34997,
    "yaw": 635.0937
},
{
    "event": "PlayerPickupItemEvent",
    "material": "COBBLESTONE",
    "var": 1
},
{
    "event": "PlayerInteractEvent",
    "blockFace": "SOUTH",
    "blockX": 527,
    "blockY": 75,
    "blockZ": 1023
    "material": "FURNACE",
    "enum": "RIGHT_CLICK_BLOCK"
},
{
    "event": "InventoryOpenEvent"
}
```
- **time** represents the time in milliseconds when the event occurred on the server.
- **player** identifies the player triggering the event with an integer calculated by a irreversible hash function.
- **worldUUID** identifies the world in which the event occurred.

The generic table contains the rows **enum** and **var** which represent different things depending on the event. In case of **PlayerPickupItemEvent**, **var** represents the number of items that have been picked up. **enum** can contain a string where **var** is limited to integers. All string values seen in Fig. A.1 are reduced to numbers by using a lookup table inside the database. This drastically reduces the size of the database since the strings are represented by enumerations inside the game and are repeated very frequently.

Representing different kinds of events by small set of data structures helped reducing database size and code complexity. However, many compromises had to be found when the number of interesting attributes would not fit the existing structures. Using a database allows us to use SQL which is great for selecting subsets or do simple data aggregation or statistics. But storing the data in a less structured format would have given us more flexibility and might have ultimately led to covering more data.

In addition to writing all received data to a SQLite database, we kept daily logs of the raw JSON data sent to the server as a backup. When compressing those log files, the resulting files had only half the size of the database. Using compressed JSON files as data backend would therefore not only allow greater flexibility, but also save space. The logs could still be fed into a SQL database for easier access afterwards.
A. Communication and Data Storage
Live Server Stats

The companion website for our Minecraft server features live stats (Fig. B.1) and an automatically updated heat map (Fig. 6.3). We wanted to motivate people to play on our server by allowing them to affect the numbers and the map in realtime. The stats seem trivial, but the kind of heat map we have is not yet common on other servers. Additional graphs, interactivity and data about development over time could be added to make the site more appealing to both players and server admins. The site could then be provided as a service to server admins in return for them installing our data collection plugin.

Most stats can be calculated by a simple `count(*)` request against our SQL database. Calculating the play time, walking distance and the heat map on the other hand had to include every entry from the move table. After recording 14 days of gameplay, the database had about 15 million move entries and over 6 million other events. The total size of the SQLite database file was around 1 GB. At that time, even basic count statements required several seconds to execute.

We calculated the play time by checking whether a player was moving. Players not changing their position for more then 20 seconds are considered idle and their presence is not added to the total play time. When creating the heat map we ignored players idle for more than 1 second. For the move distances we ignored distances greater than 5 blocks within one tick (20 ms). That way we are able to filter teleportation e.g. when dying and respawning.

All those calculations require traversing the giant move table which using our Python or PHP scripts took several minutes. Using caches we were able to update our results in seconds. This way the stats can be updated every time the Epilog plugin sends new data (at most every 10 seconds). A couple of seconds every 10 seconds is still a lot of computing time, especially on a web server running websites for the whole department. So we decided to run the scripts on randomly selected workstations.
B. Live Server Stats

Figure B.1.: Live server stats section of the companion website
Bibliography


