MC-Saar-Instruct: a Platform for Minecraft Instruction Giving Agents

Arne Köhn and **Julia Wichlacz** and **Christine Schäfer**
Álvaro Torralba and **Jörg Hoffmann** and **Alexander Koller**
koehn@coli.uni-saarland.de, wichlacz@cs.uni-saarland.de, cschaef@coli.uni-saarland.de,
torralba@cs.uni-saarland.de, hoffmann@cs.uni-saarland.de, koller@coli.uni-saarland.de
Saarland University

Abstract

We present a comprehensive platform to run human-computer experiments where an agent instructs a human in Minecraft, a 3D blocksworld environment. This platform enables comparisons between different agents by matching users to agents. It performs extensive logging and takes care of all boilerplate, allowing to easily incorporate new agents to evaluate them. Our environment is prepared to evaluate any kind of instruction giving system, recording the interaction and all actions of the user. We provide example architects, a Wizard-of-Oz architect and set-up scripts to automatically download, build and start the platform.

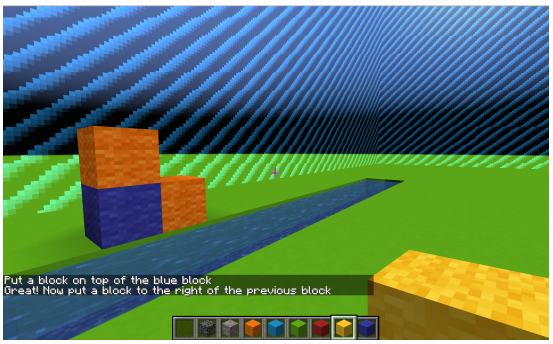

Figure 1: Example of instructions provided to a user. New instructions appear at the bottom of the chat text and old ones fade out after five seconds. In this case, the world is initialized with a tiny river; the stripes are the barriers for the user.

1 Introduction

Collaborative human-computer interaction can occur in different environments. While interaction in the physical world is often a desirable goal, it places a huge burden on automatic agents as perception is a hard problem, raising the barrier of setting up such experiments significantly. On the other end, interactions on a custom-built platform may be a good fit to explore specific phenomena, but they do not scale easily to different or complex problems. A good example for a custom-built virtual 3D world is the GIVE challenge, where an instruction system must guide a player to press a specific sequence of buttons in a 3D environment while avoiding to step into traps (Byron et al., 2009; Striegnitz et al., 2011). We instead use a *general-purpose* 3D environment.

We release an experimentation platform based on Minecraft (see Figure 1). Minecraft is a game in which the players are situated in a 3D world, which mainly consists of blocks. The game can either be played locally as a single-player game or one can join an online server and play with others. The players can move around, place and remove blocks, and even craft new blocks or items. As such, Minecraft can be seen as a classic blocksworld that can be scaled up a lot in complexity: Blocks can have different types (wood, earth, stone, glass, lamps, ...), they can be combined into high-level objects, and special blocks even enable building circuits, resulting in Turing-complete machinery. Minecraft contains different game modes: a survival mode, which focuses on exploration and survival in the game world, and the creative mode, focusing on building. We make use of the creative mode.

This feature-richness makes Minecraft a perfect environment for the evaluation of all kinds of intelligent agents (Johnson et al., 2016), from reinforcement learning agents (Guss et al., 2019), to instruction receiving (Szlam et al., 2019) and instruction giving assistants (Narayan-Chen et al., 2019). Its popularity (Minecraft is the most sold game of all time), together with the client-server architecture make Minecraft a tool well-suited for crowd-sourcing with volunteers from all over the world. Moreover, there are tons of instruction videos for Minecraft on the internet which could be used as auxiliary datasets for offline instruction giving. This addresses several of the limitations

Proceedings of the 21th Annual Meeting of the Special Interest Group on Discourse and Dialogue, pages 215 - 218

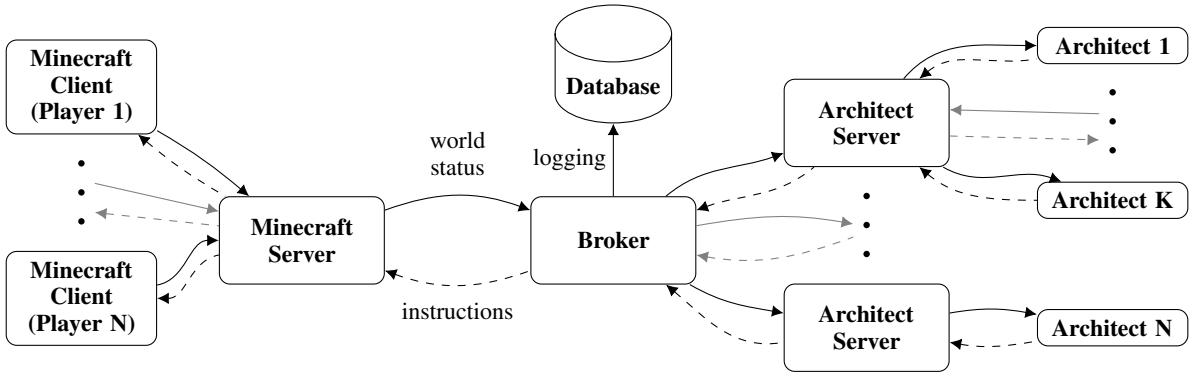

Figure 2: Overview of the services in MC-Saar-Instruct. Updates on the world state are passed along the full lines, instructions are forwarded along the dashed lines.

that previous frameworks like GIVE had: attracting an even larger number of users for the experiments, being more engaging, and allowing for a variety of experiments of increasing complexity.

The platform presented here makes it easy to set up and run instruction giving experiments in Minecraft. In our research, we focus on instructing the user to build complex objects (Wichlacz et al., 2019; Köhn and Koller, 2019), but our platform can easily be used for other generation tasks.

2 System Overview

MC-Saar-Instruct is implemented as a distributed platform which is shown in Figure 2. It consists of the following components, which can each run on their own server:

- The Minecraft server accepts connections from users.
- The Broker decides which scenario will be played by the user, tells the Minecraft server how to initialize the user's world, pairs the user with an architect and logs all interactions.
- The Architect is the agent with which the user interacts. It receives status updates of the world through the broker and sends natural language instructions back.

While there is only one Minecraft server and one broker, there can be several different kinds of architects, each hosted by its own Architect Server. All interactions between these components are handled using the grpc library,[1] abstracting away the low-level networking and providing a succinct and type-save remote procedure call (RPC) interface.

[1] https://grpc.io/

We provide example Architect Servers in Java, but they can be written in any language with grpc bindings, such as Python, Go, and many more.

2.1 The Minecraft Server

In contrast to other experimentation systems, such as Johnson et al. (2016) (who modify the Minecraft client) or Szlam et al. (2019) (who use the third-party server Cuberite), we make use of the official Minecraft server, which means that users can use an unmodified up-to-date Minecraft client. Experiments can also make use of all features introduced by new Minecraft releases, if they wish. All functionality in Minecraft, including building Turing-complete apparatuses, can be used.

Upon entering the server, each player is teleported into their own world, which is automatically set up to reflect the start state of the scenario selected by the broker (see Figure 1). All interaction between players is inhibited and all changes made by players are reset once they disconnect. Movement is restricted to a square area and players cannot remove the bottom-most layer of the world and fall into the void. World changes not caused by the player (e. g. weather, time) are disabled. The Minecraft Server runs in creative mode so players have infinite access to building blocks and no decreasing hunger or health bars.

Every 100ms, the server sends the current player position and orientation to the broker. It also sends updates whenever the state of the world changes, i. e. whenever a block is placed or destroyed.

Whenever the architect or the broker sends a message to a user, it is shown as a standard chat message (see Figure 1). Players can also send chat messages to the broker. This can be used for responses in experiment surveys (see Section 5) or for

an architect that can handle clarification questions.

Because all modifications are implemented in a server plugin, players can connect with an unmodified Minecraft client over the internet.

2.2 The Broker

The broker is the centerpiece of the whole system. It connects to all Architect Servers and provides an RPC interface for the Minecraft server. Whenever a player joins the Minecraft server, the broker gets a message and decides which scenario should be played and what kind of architect the user should be paired with. It then sends a request to the corresponding Architect Server to initialize a new architect and matches that architect to the player. Other than these decisions, the broker is mostly passive. All communication between architect and player is routed over the broker. The broker logs all messages to a database, i. e. block additions and deletions, text messages sent to and from the user and position and camera orientation updates. It also logs the start and end times of experiments and each questionnaire.

The broker provides a web interface to monitor the experiments. It shows the status of the newest experiments and can show a complete list of all database records from a specific game. An in-memory database can be used for development purposes so that no local database needs to be set up and the database is clean on every start.

2.3 The Architect

The architect generates the instructions for the users. Each kind of architect is hosted by an Architect Server. Every time an experiment is supposed to start with this type of architect, the Architect Server instantiates a new architect. The server keeps track of which architect is connected to which game and forwards messages from the broker to the correct architect.

The architect is what a researcher developing and evaluating a new instruction-giving agent needs to implement, using e.g. our high-level Java API. The Architect Server, which manages different architects, can then be reused without changes. Architects could also be implemented in other language with grpc bindings; this would then require reimplementing the Architect Server in the new language.

In our Java API, an architect must implement four functions (see Figure 3): one is called when a block is placed, one when a block is destroyed, one for every update of the position and orientation

```
handleStatusInformation(StatusM);
handleBlockPlaced(BlockPlacedM);
handleBlockDestroyed(BlockDestroyedM);
String getArchitectInformation();
```

Figure 3: Interface to implement for a new architect. The base class provides a method to send text messages.

```
3,2,3,BLUE_WOOL
1,1,4,WATER
2,1,4,WATER
...
```

Figure 4: Excerpt from world file for Figure 1. Each line has the X, Y and Z coordinates plus the block type.

of the player and one to obtain the name of the architect. The architect can then send a string to the user at any time, to be displayed in their Minecraft client. A basic architect can be implemented in 80 lines of Java code.

The architect also determines when the player has reached the objective, as it is the only component keeping track of the state of the game. This design means that all experiment-specific logic is encapsulated in the architect and both broker and Minecraft server can always stay unchanged.

3 Defining and Running Experiments

An experiment is defined by two components: the scenarios that the players are supposed to work on and the architects that should be evaluated.

A scenario consists of a definition of an initial state of the world and architect-specific information instructing the architect of the goal. The initial world state is given by a list of blocks with their location and type (see Figure 4). Each scenario is identified by a unique name. We use a shared dependency for all components that contains the necessary descriptions of the world state when starting a scenario as well as the scenario-specific data for the architects, ensuring that the architects and the Minecraft server use the same initial setup.

4 Wizard-of-Oz Architect

We also ship a Wizard-of-Oz architect (woz) to perform human-human interaction experiments. This architect runs in a second Minecraft server where only one player may log in. That player can neither move nor place or destroy blocks. Once this architect is paired with a player by the broker, the viewpoint of the woz player is synchronized with the player, i. e. the woz player always sees exactly

what the player sees. The woz player may send text messages and these are forwarded as instructions in the same manner as those generated by an automatic agent.

We conducted initial experiments with spoken interaction and noticed that the instruction givers used patterns only possible with spoken interaction such as exactly timing single words to the instruction follower's actions and self-correction. The text-based Wizard-of-Oz setup on the other hand mirrors the setup with an automatic architect as closely as possible.

5 Post-experiment Questionnaires

After finishing an experiment, the participants fill out a questionnaire using the in-game chat. Once the architect determines that a game is over (hopefully in a successful way), the broker takes over the communication channel and asks the user a series of configurable questions. The questions and answers to this post-experiment questionnaire are logged to the database.

The in-game questionnaire allows to keep all interaction with the experiment platform inside a single medium by removing the need to e. g. open a website. It also ensures that the questionnaires and experiment data are always correctly matched. Finally, the questionnaire mechanism can be used for fraud prevention (Villalba, 2019).

6 Conclusions

We introduced a system for researching situated human-computer dialogue in the Minecraft domain. While primarily focused on instruction giving, it can potentially also be used for two-way text interaction. The framework abstracts away from most of the low-level system, providing a clean and easy to use interface for implementing instruction givers. The system also takes care of matching study participants with different architects and logging of all interactions. We ship several example architects, including a Wizard of Oz architect.

We plan to implement a replay viewer which streams the previously recorded actions by a participant to a Minecraft server. All necessary data is already being stored in the database.

MC-Saar-Instruct as well as scripts to automatically download, build and run specific versions of it for reproducible experiments are available from `https://minecraft-saar.github.io`.

Acknowledgements We thank the reviewers for their comments. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 232722074 – SFB 1102.

References

Donna Byron, Alexander Koller, Kristina Striegnitz, Justine Cassell, Robert Dale, Johanna D. Moore, and Jon Oberlander. 2009. Report on the first NLG challenge on generating instructions in virtual environments (GIVE). In *ENLG 2009 - Proceedings of the 12th European Workshop on Natural Language Generation, March 30-31, 2009, Athens, Greece*, pages 165–173. The Association for Computer Linguistics.

William H. Guss, Brandon Houghton, Nicholay Topin, Phillip Wang, Cayden Codel, Manuela Veloso, and Ruslan Salakhutdinov. 2019. MineRL: A large-scale dataset of Minecraft demonstrations. *International Joint Conference on Artificial Intelligence (IJCAI)*.

Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. 2016. The Malmo platform for artificial intelligence experimentation. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4246–4247.

Arne Köhn and Alexander Koller. 2019. Talking about what is not there: Generating indefinite referring expressions in Minecraft. In *Proceedings of the 12th International Conference on Natural Language Generation*, pages 1–10, Tokyo, Japan. Association for Computational Linguistics.

Anjali Narayan-Chen, Prashant Jayannavar, and Julia Hockenmaier. 2019. Collaborative dialogue in Minecraft. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 5405–5415, Florence, Italy. Association for Computational Linguistics.

Kristina Striegnitz, Alexandre Denis, Andrew Gargett, Konstantina Garoufi, Alexander Koller, and Mariët Theune. 2011. Report on the second second challenge on generating instructions in virtual environments (GIVE-2.5). In *ENLG 2011 - Proceedings of the 13th European Workshop on Natural Language Generation*, pages 270–279. The Association for Computer Linguistics.

Arthur Szlam, Jonathan Gray, Kavya Srinet, Yacine Jernite, Armand Joulin, Gabriel Synnaeve, Douwe Kiela, Haonan Yu, Zhuoyuan Chen, Siddharth Goyal, et al. 2019. Why build an assistant in Minecraft? *arXiv preprint arXiv:1907.09273*.

Martin Villalba. 2019. *Prediction, detection, and correction of misunderstandings in interactive tasks*. Ph.D. thesis, Saarland University.

Julia Wichlacz, Alvaro Torralba, and Jörg Hoffmann. 2019. Construction-planning models in Minecraft. In *Proceedings of the ICAPS Workshop on Hierarchical Planning*, pages 1–5.