Report NDVW

Anicet Nougaret, Jens De Bock, Maciej Chylak, Jan Bausch

Barcelona - Spain

1. Introduction

The final project will be a simulation of a 3D procedural world featuring two competing Artificial Intelligence (AI) agents, one having to gather as many resources as possible, hidden in the world, before the end of a timer, the second one having to touch the first agent in order to kill him. See 1.

The world, procedurally generated, is a volume filled entirely with individual cubes with a volume of roughly one meter cube each. The cubes, or "blocks" may initially be of 3 types: "void" (empty space), "solid", and "solid precious". The generated landscape made out of these blocks may feature little mounds, holes, and other scaled-down geological-like formations, with "solid precious" blocks often hidden in small quantity inside larger volumes of "solid" blocks.

The goal of this project is to train a "player" AI agent sharing the world with an "enemy" AI agent. The player will be capable of playing the game and collecting as many precious blocks as possible while avoiding death from the enemy.

2. Related work

The game world and the enemy agent take most inspiration from Minecraft, the famous sandbox game by Mojang Studios released in 2011 [**?**]. Especially the enemy agent's behavior can be related to the Creeper enemy in Minecraft, which follows you on sight and on sound trigger, and is able to kill the player in one shot. Although it is unclear if any of Minecraft's enemy agents are implemented using state machines, it is clear it could be a good and simple

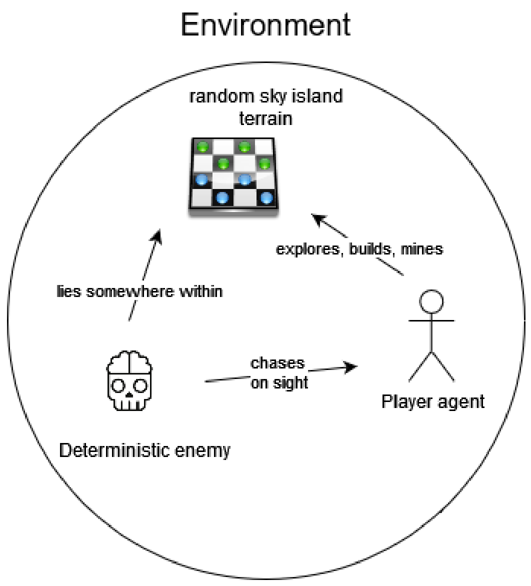


Figure 1: Concept of the game

way to do it, as most behaviors of entities in the game observe a “sense” then “move” then “act/attack” pattern.

The player agent implemented using Deep Reinforcement Learning (DRL) is more of an experimental approach that is not very suitable for enemy agents in game, and more suitable to research and development in the area of general AI. It is based on technologies developed for more than half a century to solve real-life problems, like self-driving cars, autonomous robots, recommender systems, fraud recognition [?]. Such AIs are being developed for research on many games, as ways to experiment with exotic or multi-agent environments, which would be more expensive to experiment using real robots in a real life [?]. Learning to interact with a sandbox environment like in our game is a prime example of an experiment that would be costly to conduct in real life directly.

Famous examples of DRL are works done by the DeepMind researchers. For instance, on competitive AIs for StarCraft II, the real-time strategy game, which is one of the first successful attempt at making a competitive agent in a complex, real-time, partial-information game using Deep Reinforcement Learning [?].

We also take inspiration on their famous work on Quake III, the team-based first-person shooter, where agents are trained on many randomly generated environments at the same time, an idea we would like to try [**?**]. As well as their use of Convolutional Neural Networks (CNN) in most of their work to allow the agent to learn to use the camera input directly as its main source of information.

It is important to note that the literature on training agents to play 3D games using a subjective camera view, or other quite challenging high-dimensional situations, is quite abundant since the 2015's and famous demonstrations by OpenAI and Google DeepMind, but they often relied on scaling and parallelizing dramatically on modern hardware, Reinforcement Learning algorithms and Deep Neural Networks related techniques introduced since the late 1980s by many independent researchers.

3. Proposal

3.1. Description of the agents

The two agents are roughly 1.5 meters tall, 0.9 meter wide solid volumes, able to move in all cardinal directions at a human-like speed, and also to jump roughly 1 meter high and 3 meters far. They appear at the beginning of the simulation at two random positions within two dedicated quadrants of the game environment, and always on top of a solid block, given there is enough free space above for them to stand. This is needed because the "enemy" could spawn on top of the player, and ending the game prematurely. Increasing the amount of enemy agents, or "Creepers", as we may call them in the following report, is possible until the amount hits 3, because every Creeper will have its dedicated quadrant to spawn in.

The player agent, is able to break any solid block or solid precious block, store them, and place them back next to any other block. All solid blocks in the player's inventory must be placed back into the world before solid precious blocks can be placed back as well. The player has an infinite space inventory consisting of a simple counter for each kind of block. By storing solid precious blocks, the player earns points. It is the only way to earn game points. If the game timer ends, it wins and gets its score, if it dies, either by jumping into the void and reaching a coordinate $y < 0$, or by colliding with the Creeper, it loses and gets a score of -1. The player agent will follow a stochastic, black-box, Deep Reinforcement Learning-based behavior, described in Section 3.2.2.

The enemy agent, or "Creeper", can kill the player by simply colliding with it. The enemy wins if the player dies. The Creeper will follow a programmatic behavior described in Section 3.2.1.

3.2. AI Design

3.2.1. Long form specification of the enemy agent

The enemy is a programmatic AI with different simple behaviors. In its initial state, it will stay stationary and have physical collision with the world environment. A hearing sense is simulated with a radius of five meter around the Creeper. If the distance to the player is closer than this radius, the Creeper will detect the player and turn around to face him.

Next, the Creeper will start walking in the direction of the player. To avoid him getting stuck in the world, it jumps at random time intervals. The player will have the chance to out-run the Creeper by getting outside its hearing radius again.

Otherwise, the Creeper will get closer and closer and, triggered by a collision between the Creeper and player, will explode and kill the player.

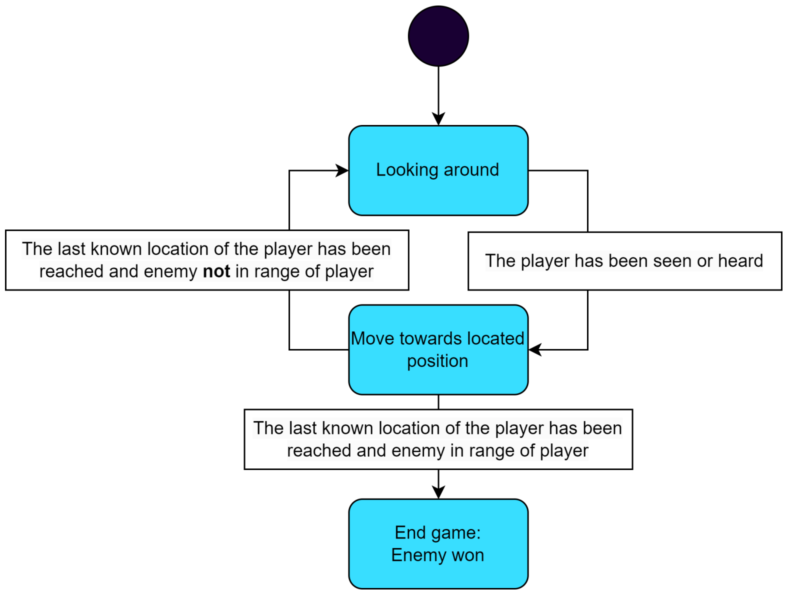

Looking around
The last known location of the player has been reached and enemy **not** in range of player
The player has been seen or heard
Move towards located position
The last known location of the player has been reached and enemy in range of player
End game: Enemy won

Figure 2: State transition diagram of the enemy agent

3.2.2. Long form specification of the player agent

The player agent design is trying to solve a Partially Observable Markov Decision Process using Deep Reinforcement Learning. Two model were developed: a "World-aware" model, and a "From-scratch" model.

At each game frame, or time step t (every 0.1 seconds, as this represents a realistic lower bound for human reaction time in video-games), both of these two models receive a partial observation $\hat{s}_t$ of the world's state s_t. This partial observation would always include the inventory counts: sb_t for solid blocks, spb_t for solid precious blocks.

Then, in the case of the From-scratch model, it also consists of a 64×64px RGB frame captured from the player's front-facing camera: $\phi_t = \{r_1, ..., r_{64\times64}, g_1, ..., g_{64\times64}, b_1, ..., b_{64\times64}\}$, from which the From-scratch model has the difficult and totally indirect challenge of learning to extract useful information, such as the position of the precious blocks and of the creeper relative to him.

$\hat{s}_t^{\text{(from-scratch)}} = \{\text{sb}_t, \text{spb}_t, r_1, ..., r_{64\times64}, g_1, ..., g_{64\times64}, b_1, ..., b_{64\times64}\}$

And in the case of the World-aware model, it consists of data from 16x16 different ray-casts, spanning the full field-of-view of the camera. Each ray-cast either collides with a block or with nothing. In case it collides with a solid block, it will yield a value of 1, if it is with a solid precious block, a value of 2, and if nothing is hit, a value of 0. It will also yield the distance of the collision, or the maximal distance 10 if nothing was hit. Since the rays are being cast through a grid at the position of the camera, they can be rebuilt into a raster picture. Hence, two channels, Red and Green, are used to store the ray results: $\phi_t = \{\text{block}_1, ..., \text{block}_{16\times16}, \text{dist}_1, ..., \text{dist}_{16\times16}\}$. Additionally, because the rays don't collide with the Creeper, its relative position to the player $\text{pos}_r = (x_r, y_r, z_r)$ is added to $\hat{s}_t$.

$$\hat{s}_t^{(\text{world-aware})} = \{\text{sb}_t, \text{spb}_t, x_r, y_r, z_r, \text{block}_1, ..., \text{block}_{16\times16}, \text{dist}_1, ..., \text{dist}_{16\times16}\}$$

The World-aware model is not totally omniscient, and is not even so much more informed than the From-scratch model, as both see the same area of the world (except for the Creeper's position). Indeed, the World-aware model just enjoys the same information as an idealized From-scratch model, perfectly tuned to extract information from the observation, would be able to guess.

The agent, regardless of the model, has a policy $\pi(s_t)$ that dictates what action a_t to take amongst 11 actions: go left, right, backward, forward, jump, rotate its camera up, down, left, right, break a block in front or place a block in front.

Its actions impact the world's state, therefore it receives back the next observation $\hat{s}_{t+1}$ and a reward R_t, mostly depending on the world's state, in order to guide the agent towards its objective.

The reward is designed and tweaked as needed during the development and training phase, but it has to reward very positively for precious blocks, and very negatively for death. No smaller variations of the reward upon other events as to nudge the AI into desired directions (exploration, creativity, anticipation of the enemy. . .) is added. Although it is a proven technique, we believe it is more challenging and interesting to let it converge towards its own strategies without too much nudging. Learning with the most sparse and important rewards, can be seen as one of the main engineering goal, intellectual appeal and difficulty of Reinforcement Learning.

The goal of the learning agent is to converge towards π^*, the optimal policy, which selects actions in order to reach states that maximize the expected sum of discounted future rewards, or "return", $V_\pi^*(s_t) = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k}]$.

3.2.3. Algorithms explored for the player agent

Many Deep Learning Algorithms and variations exist to converge towards this optimal policy. Our intent was to try as many as possible and to use little RL-specific frameworks as to force ourselves to really understand things by re-implementing them.

We first focused on the harder From-scratch AI, and started simple with classic ϵ-greedy Deep Q-Learning. Then, as this wouldn't work well, we made the approach more involved, with Double Deep Q-Learning, n-step Deep Q-Learning, Memory Replay, Prioritized Memory Replay, until we didn't have time to explore more techniques, such as Policy Gradient or Actor-Critic approaches. Which we would love to explore in the future.

It is important to note that our approach is model-free. But most state-of-the-art approaches in the literature learn a Hidden Markov process of the environment, and plan upon it something like Monte Carlo tree search. This would also have been interesting to do, but was judged too difficult for the time we have.

We also gradually tried other approaches than classic ϵ-greediness with ϵ decay for solving the exploration/exploitation tradeoff, such as oscillating ϵ, or by using radically different techniques such as Noisy Linear layers: fully-connected layers which inject noise into their outputs as to explore the action-space, and which learn the parameters of the Gaussian noise they inject.

Some related works learn their own reward, or their own part of the reward, as to converge towards optimal intrinsic rewards, nudging exploration, risk, creativity..., which is interesting, but which we didn't have time to try.

We also had to deal with the partially observable nature of the process, by experimenting with various memory units (RNN, LSTMs. . .).

Once we had no more time to implement a new, more sophisticated approach, we first trained the From-scratch model on the final approach, and then trained the World-aware model with the same approach.

Seeing the From-scratch model never really succeed but slightly improve all the time, motivated us to go as far as possible, as probably no approach would have been enough for the From-scratch model to perform well after only a few days of training, and all of them would have trained the World-aware one in a reasonable time. This way, we explored a lot and learned a lot.

3.2.4. Final algorithm and architectures

The final RL approach used was a double n-step Deep Q-learning approach, as it would converge the fastest in our experience.

$\epsilon-$greediness with decay was not relied on, only noise-injection from Noisy Linear layers sufficed for exploration in our case.

Any kind of Memory Replay was at first prohibitively costly, as it consumed a lot of RAM. But at some point we upgraded our hardware and it became cheap. Therefore, we would use Prioritized Memory Replay. It would store up to 450 (From-scratch) or 1800 (World-aware) past played episodes (or "trajectories"), each consisting of 300 consecutive frames. The ranking criteria for the trajectories consisted of the sum of absolute values of all rewards in the episode. The model would play all incoming trajectories from the Unity game, but it would execute the costly learning operations on the best ranked recorded episodes only, reducing rewards scarcity in general by roughly a factor of 4, therefore making the learning phase 4 times more informative for the model.

Both models had similar architectures, with Noisy Linear layers of the same size and an identical LSTM layer. But they had different CNN architectures for pattern extraction from the visual inputs:

Certainly that the From-scratch model, having to learn to see in 3D, with a moving camera, benefits from a solid CNN architecture, battle-tested in image classification tasks, instead of the classic 2-layers CNN used in toy models for 2D Atari games learning. Which is why its CNN featured many Residual blocks and Down scaling Residual blocks inspired from the previously state-of-the-art image classification ResNet-50 architecture 5. This probably dramatically increased the model's parameters count, but we do not believe it learned anything from observations with smaller models.

On the other side, the World-aware model had a CNN architecture similar to the one used for learning 2D Atari games, with only two layers. Using a CNN for the ray results was certainly more interesting than not using them,

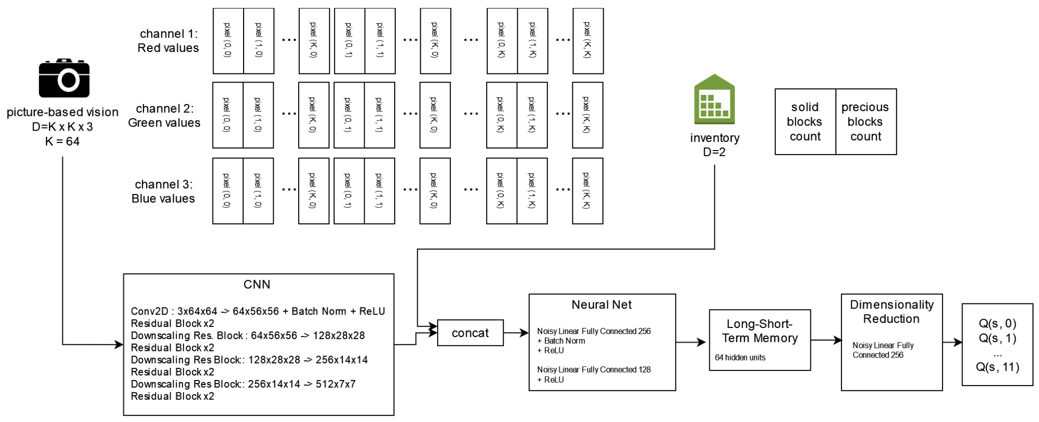


Figure 3: Architecture of the From-scratch model

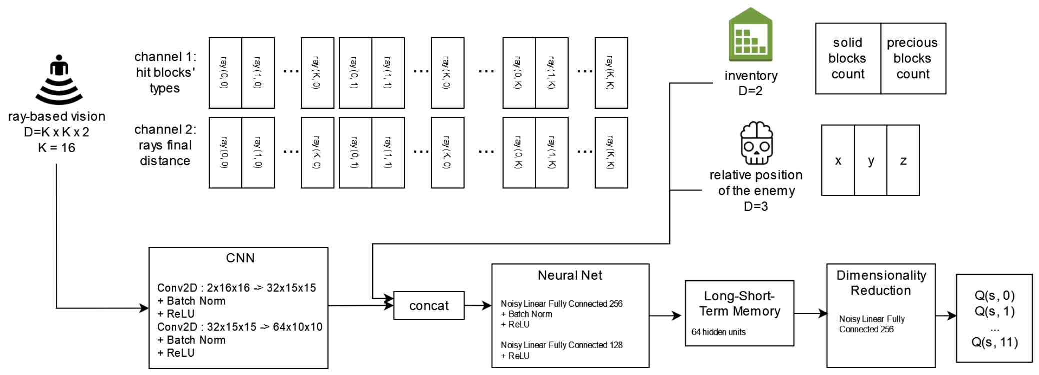


Figure 4: Architecture of the World-aware model

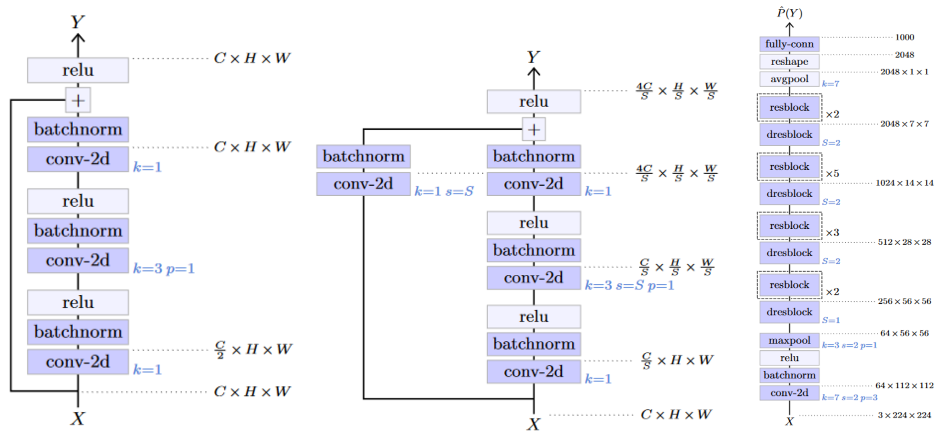


Figure 5: Residual block, Down scaling Residual block and ResNet-50 architecture. Diagrams by François Chollet under Creative Commons BY-NC-SA 4.0 International License.

although they are not exactly pictures, they still have strong local spatial correlations, which is what CNNs excel at learning.

3.3. Software architecture and parallel training setup

Since the beginning, we had to design both a game and a Deep Reinforcement Learning platform, as we always intended to create our own alternative to Unity's ML agent. Therefore, the game architecture had the unusual requirement to be easily replicable amongst multiple environment instances and most component interactions to be scripted rather than handled in the editor as to allow ”hijacking” the game totally when the AI is learning. This made implementing the features more tricky, with significant effort put into gameplay performance, avoiding memory leaks, and other things that would be game breaking issues once you start duplicating the environments massively. But it also made the architecture more organized by necessity.

There are two modes for the game's architecture:

- ”play” mode is for testing and playing the game directly.
- ”remote-controlled” mode, is for letting the game be controlled remotely by the software created for training and running the Deep Learning model.

3.3.1. Architecture and module interactions of the game in "play" mode

See figure 6 for the architecture of an individual environment and for the explanation of the modules' interactions between one-another in "play" mode.

See figure 7 for the high-level architecture and explanation of the full scene in "play" mode.

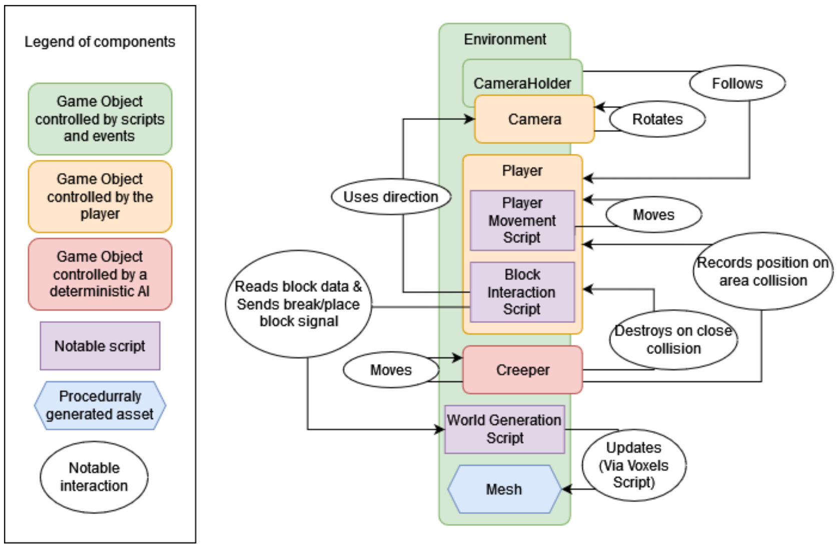

Legend of components
Game Object controlled by scripts and events
Game Object controlled by the player
Game Object controlled by a deterministic AI
Notable script
Procedurraly generated asset
Notable interaction
Environment
CameraHolder
Follows
Camera
Rotates
Player
Uses direction
Player Movement Script
Moves
Records position on area collision
Reads block data & Sends break/place block signal
Block Interaction Script
Destroys on close collision
Moves
Creeper
World Generation Script
Updates (Via Voxels Script)
Mesh

Figure 6: Zoom on the architecture of a single environment in "play" mode.

3.3.2. Architecture and module interactions of the game in "remote controlled" mode

The "remote controlled" mode is relevant for training AI models on the game. It is a custom replacement for Unity's ML agents, enabling synchronizing and sending states to the model via the network and executing the received actions.

The Unity game becomes a TCP server, and the model a TCP client. They communicate with one another using a custom protocol. See 8 for a high-level view of multiple environments in "remote controlled" mode. See 9 for a detailed view of the architecture of multiple environments in "remote controlled" mode.

In "remote controlled" mode, which can be enabled on the Multiple Environments Initialization script, two additional scripts are attached to the Simulation Game Object: a Server script and a Conductor script. The latter

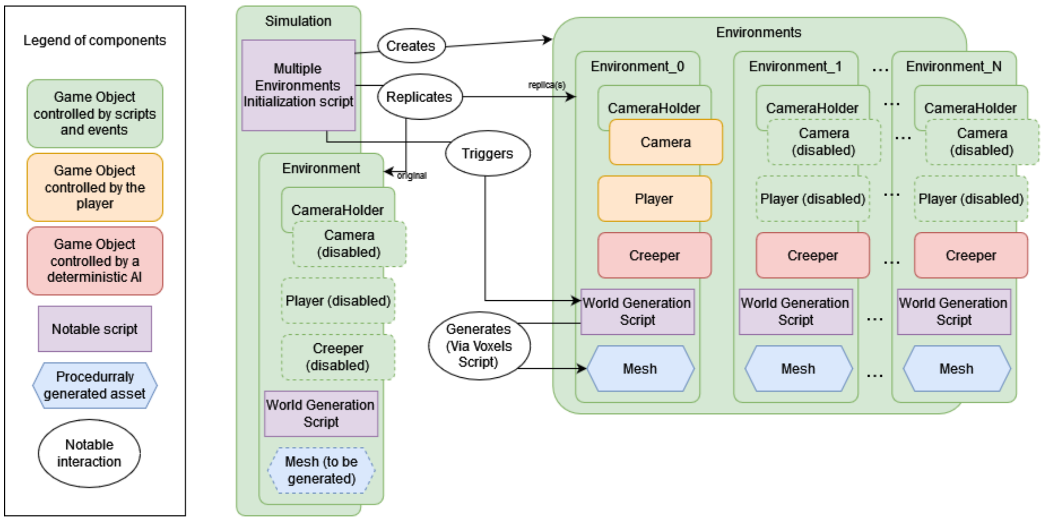


Figure 7: Architecture of the scene in ”play” mode. At first, only the Simulation Game Object exists, then it instantiates multiple Environment Game Objects within the Environments Game Object, and triggers their world generation procedures. In ”play” mode, although it may seem unnecessary, spawning multiple worlds is kept for debugging reasons, but the other players and enemies are deactivated to save performances.

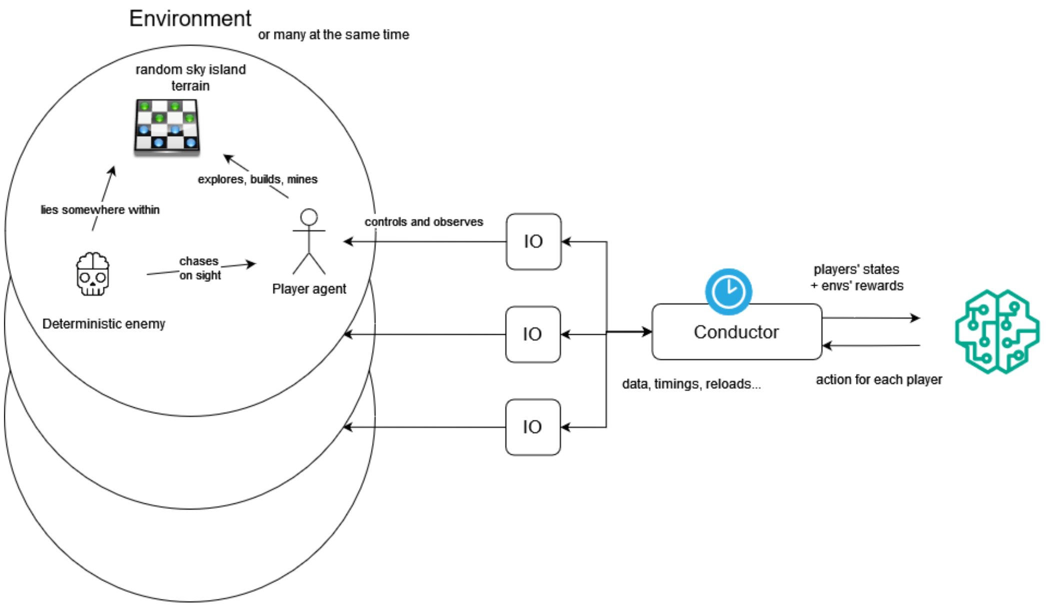


Figure 8: High-level schema of a remote controlled Environment

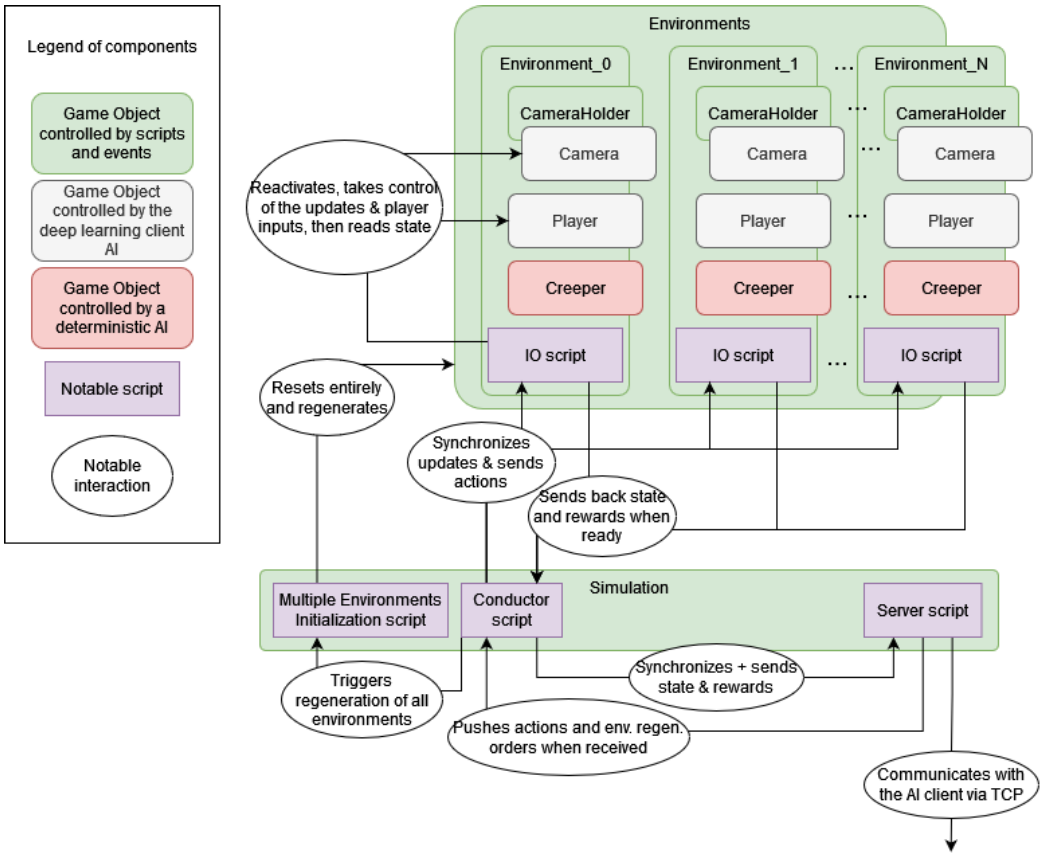

Legend of components
Game Object controlled by scripts and events
Game Object controlled by the deep learning client AI
Game Object controlled by a deterministic AI
Notable script
Notable interaction
Environments
Environment_0
Environment_1
Environment_N
CameraHolder
Camera
Player
Creeper
IO script
Reactivates, takes control of the updates & player inputs, then reads state
Resets entirely and regenerates
Synchronizes updates & sends actions
Sends back state and rewards when ready
Simulation
Multiple Environments Initialization script
Conductor script
Server script
Triggers regeneration of all environments
Synchronizes + sends state & rewards
Pushes actions and env. regen. orders when received
Communicates with the AI client via TCP

Figure 9: Detailed architecture of a remote controlled Environment

blocks the initialization and starts a TCP server, waiting for client connections. Upon connection, it waits for initialization messages. Upon them arriving, it reads the message's attached seed and environments count and informs the Conductor script.

The Conductor then triggers the initialization. After all the environments are generated, it attaches an IO script to each environment which enables all the Game Objects (Player, Camera) that were disabled and turns all the relevant Game Objects into ”remote controlled” mode. This mode lets the IO script gather information and control at a distance any of these Game Objects. It also turns their updates off, as the IO is now in charge of synchronously triggering updates for them.

A synchronous loop is now established: Upon receiving a new action for an environment (e), $a_t^{(e)}$ from the Server, upon itself receiving it from the client, the Conductor triggers an update of duration $\delta t = 0.1s$ that the environment's IO script executes on the environment's state $s_{t+1}^{(e)}$. Then the Conductor gathers one time-step of game states observations $\phi_{t+1}^{(e)}$ and rewards $R_{t+1}^{(e)}$ from the environment's IO script, and sends them to the Server, which, in turn, serializes them and sends them to the client.

3.3.3. Architecture, and throughput optimization via parallelization of the full client/server training setup

For a high-level overview of the client/server training setup, see 10.

All of what we have described so far, hinted that only one instance of the game, or Server, and only one instance of the model, or Client, would be running at a time. However, this would be too slow, and underutilize our hardware. So we have multiple of them at the same time, which involves a complicated setup. Although, it is quite normal to have this kind of setup in Deep Reinforcement Learning. Indeed, training a model such as the From-scratch model requires considerable hardware and parallelization efforts as to speed up the important training times. Considerable effort has to go in optimizing the throughput, e.g. the number of trained frames per second.

Recreating Unity's ML agent was a mistake at first, but keeping it was a bet: we believed we would eventually need a very involved parallel setup for training the final model, which would be harder to do with Unity ML agents. We had no time to check again with ML agents if the bet is indeed won. At least we learned a lot, and really owned it in a way that enabled us to iterate fast.

As to maximize throughput, a tedious balance has to be kept. Ideally, parallelization must be privileged, and all the pieces of hardware, memory, CPU and GPU, must be kept at a maximum utilization, as compromises are often possible between one-another to limit individual bottlenecks.

Both the model and the client are running on the same machine. Running both on different machines was tried and is possible, but we lack machines to do that for long periods of time. The final training machine is a shared server on Windows 10 with 8 cores 2.80GHz CPU, essentially used for running the Unity games, or Servers. It also has an NVIDIA A4500 RTX GPU with

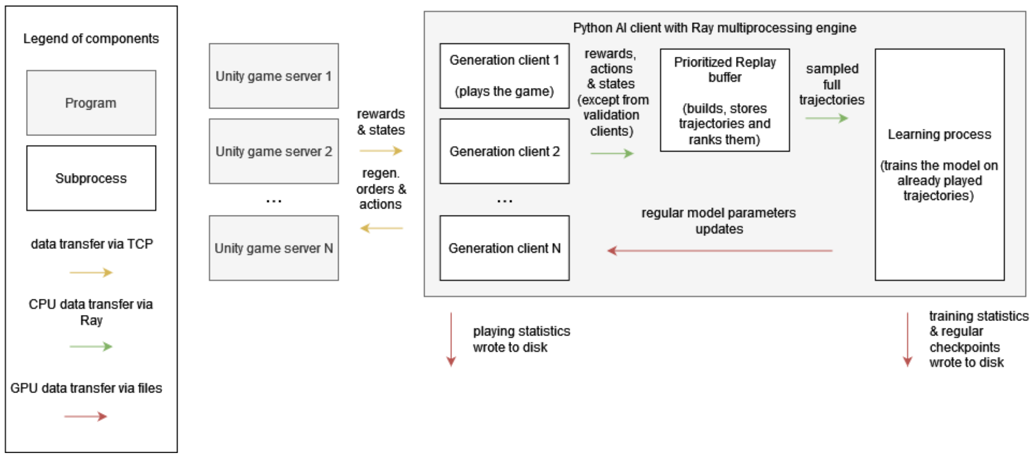

Legend of components
Program
Subprocess
data transfer via TCP
CPU data transfer via Ray
GPU data transfer via files
Unity game server 1
Unity game server 2
...
Unity game server N
rewards & states
regen. orders & actions
Python AI client with Ray multiprocessing engine
Generation client 1 (plays the game)
Generation client 2
...
Generation client N
rewards, actions & states (except from validation clients)
Prioritized Replay buffer (builds, stores trajectories and ranks them)
sampled full trajectories
Learning process (trains the model on already played trajectories)
regular model parameters updates
playing statistics wrote to disk
training statistics & regular checkpoints wrote to disk

Figure 10: Architecture and data flow of the multi-client/server training setup

20GB of VRAM, essentially used for running the model, or Clients. And finally, 28GB of RAM used heavily by both the game, the memory replay buffer and the model.

The model, running on the GPU, which is the way to go for Deep Neural Networks, has the bottleneck of long data transfers times from the CPU to the GPU. Running large parallel operations, as to utilize the GPU fully, is a way to amortize these transfer costs. For that, we must train and do inference on large enough mini-batches of observations at the same time.

The environments, simulated in Unity, don't parallelize well. We believe Unity to be the culprit here, as it seems to not be able to use more than two cores at once. So we divide our environments in multiple Unity instances, so into multiple servers. In the final setup, we had 6 instances, amongst which one was kept for validation, and on each instance, 16 simultaneous environments.

Servers compute the states and rewards for all their environments, send them to their client, then the clients sends back the action and record their unrolling trajectory in the memory replay buffer. On the meantime, a copy of the clients' model is trained on sampled trajectories from the memory replay buffer, and regularly, weights updates are sent back to update the client models. This loop is a standard DRL technique, which proved effective at fully parallelizing the tasks and using the hardware at a maximum.

3.4. Design of the game

Even though, we aim to create a game with as purpose to train an AI agent how to play a game, the operator of the game can still have some influence by changing game settings as shown in the start screen, Figure 11.

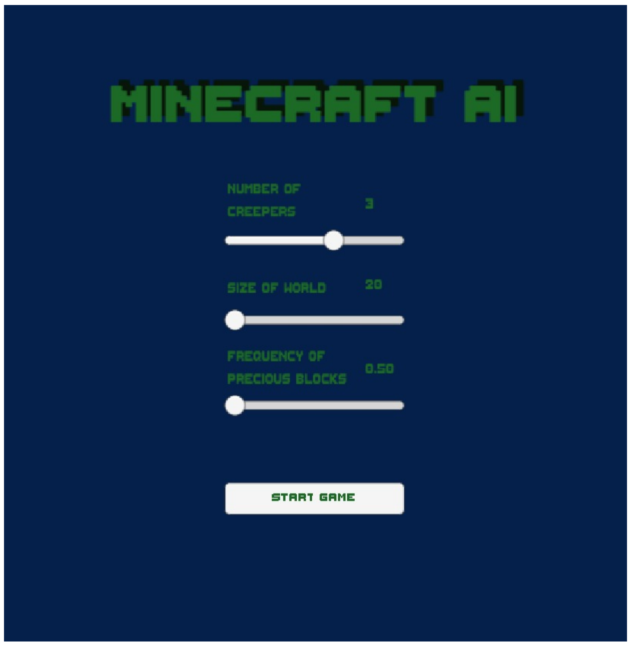


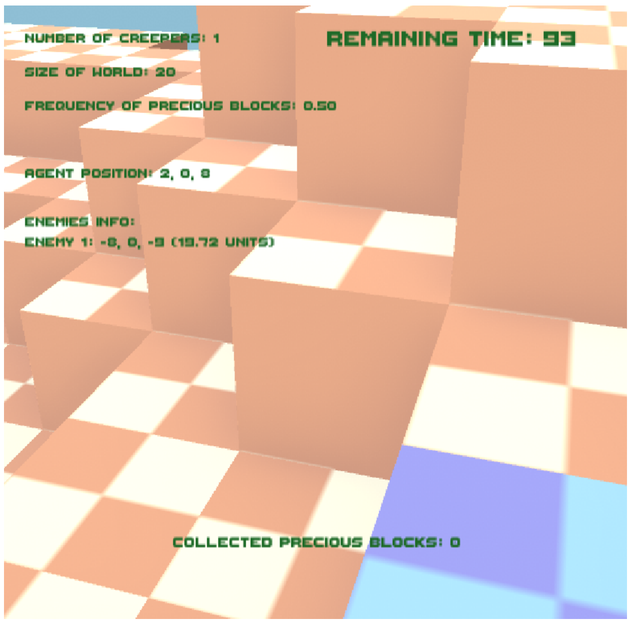
Figure 11: Start screen with options to choose the settings of the game

The following settings can be changed:

- Number of enemies [0 $\rightarrow$ 4]
- Size of world [20 $\rightarrow$ 100]
- Frequency of precious blocks [0.0 $\rightarrow$ 1.0]

These parameters will be used in the following step to load the game.

Thanks to the implemented UI, which we can preview in picture 12, we are able to see during the gameplay parameters such as the remaining time, the number of precious blocks collected and the positions of the player and the creeper.


NUMBER OF CREEPERS: 1

REMAINING TIME: 93

SIZE OF WORLD: 20

FREQUENCY OF PRECIOUS BLOCKS: 0.50

AGENT POSITION: 2, 0, 8

ENEMIES INFO:
ENEMY 1: -8, 0, -9 (19.72 UNITS)

COLLECTED PRECIOUS BLOCKS: 0

Figure 12: UI containing parameters describing the game

3.5. Methodology

The background music was generated based on the original Minecraft game music using AIVA [**?**]. The tool allows you to upload your own input files, which then serve as AI inspiration for the generated music. The Creeper has continuously-looping sound effect attached. The asset is generated by an AI Tool [**?**] that simulates a Minecraft Villager voice from any other voice input. As an input, we used a speech by US President Joe Biden.

3.5.1. Implementation of terrain generation

For the world generation, much inspiration was taken from Perlin noise, see Figure 13, which is a fast and easy way to generate fluent terrain. The world is hence random, its generation is based on a seed, so the environments will be reproducible. On top of that, it is easy to change parameters around and play with the terrain.

Once we had generated world data, we needed to render them. For this, we generate meshes from code manually, block per block. This proved to offer very poor performances though, so we optimized by rendering only visible faces, then we grouped all the meshes in the engine to further optimize. This

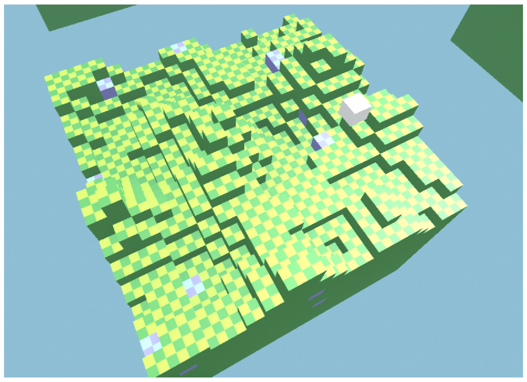


Figure 13: Example of what 20×20×20 Perlin world generation looks like

is not enough to do proper rendering, though. So we then tweak the UV's of each vertex so it maps to the correct block type texture within the blocks' atlas texture, and then we compute the faces' normal so the light interacts correctly with the mesh.

3.5.2. Implementation of enemy agent

The enemy agent is constructed using a Unity Prefab to be able to spawn multiple instances in the same game. The agent's behaviour is divided into two scripts:

Firstly, a "CreeperMovement" script is responsible for movement in the world. If the distance to the player is less than a predefined "detection-Radius", the enemy will rotate towards the player and move in his direction. Additionally, if the enemy is currently grounded on the floor, an upward force is applied in random intervals. This allows him to jump over blocks that might be in the way.

Secondly, a "CreeperExplode" script is responsible for detecting a collision between the enemy and the player. A collision causes multiple effects:

1. Animating an explosion particle system
2. Playing an explosion sound effect
3. Destroying the player game object
4. Showing the end scene

3.5.3. Implementation of player agent and Deep Learning AI

See 3.2.3 for an explanation of the algorithms implemented for the player agent, 3.3.2 for the remotely controlled environments and player agent setup, and see 3.3.3 for the communication and training setup.

Other than Unity, and our custom ML Agents clone, the Deep Learning AI was implemented using Python and the PyTorch Deep Learning framework. The parallel setup was achieved using the Ray Core library, which allows running multiple parallel Python ”agents” (not to mistake with AI ”agents”) which communicate using message-passing. Typically, each client, the memory buffer, and the learning process would all be agents and communicate via asynchronous messages.

3.5.4. Training methodology

The architectures and training setup were already explained previously, see 3.2.4 and 3.3.3.

Both the From-scratch and World-aware models were trained using the same setup, machine and almost identical hyperparameters. The World-aware model, because it has lower-dimensional inputs, was granted a larger memory replay buffer of maximum 1800 trajectories, versus 450 for the From-scratch model. Also, it had larger minibatch-size for the same reason, 512 versus 128.

First, the From-scratch model was trained with ∼6M frames from trajectories sampled from the memory replay buffer (with certainly repetition from one epoch to the other), while ∼11M frames were played and sent to the memory replay buffer for training, and ∼3M frames were just played for validation statistics but not sent. The training lasted 3 days and 6 hours.

Then, in the same way, the World-aware model was trained with ∼4.5M frames, while ∼6M frames were played and sent, and ∼2M frames were just played for validation.

3.6. Results

3.6.1. Final gameplay

This short video shows some basic elements in our game including the movement, player rotation, the breaking, and placing of blocks and the enemy agent.

3.6.2. Training results

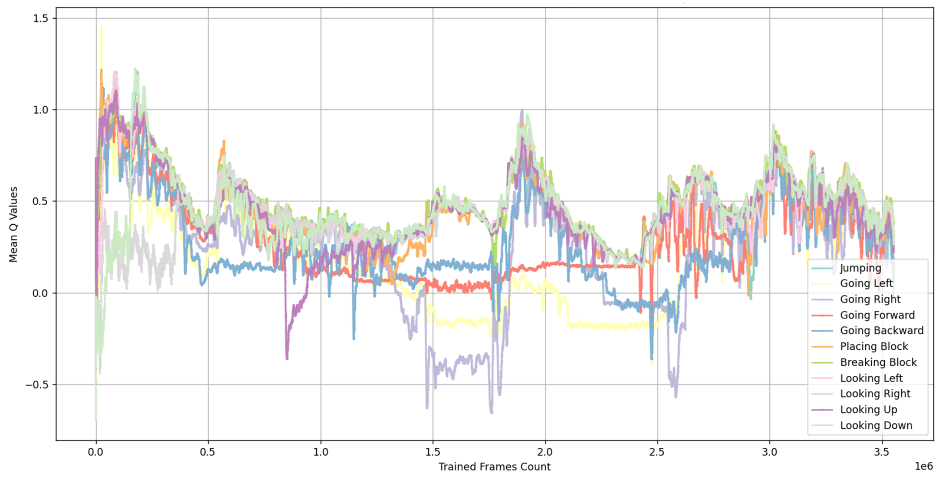
In the end, the World-aware model attains higher scores. We can see from 14b that it is learning to play better and better from its Q-values. In opposition with the same graph for the From-scratch model 14a, which shows very unstable learning and stagnating scores. Still, the World-aware model will soon reach a plateau, while not getting human-level scores (which we expect to be around 10 precious blocks per episode). Hence, the final model-free Q-learning-based approach itself may not be able to beat human players at our game.

Again, we can see the same trend difference from the validation data, which consists of the highest final rewards sum out of 16 environments over the 300 first episodes 15. This validation data reflects well our From-scratch model's inability to generalize, and our World-aware model's ability to generalize, up to some noticeable, albeit limited, skill level.

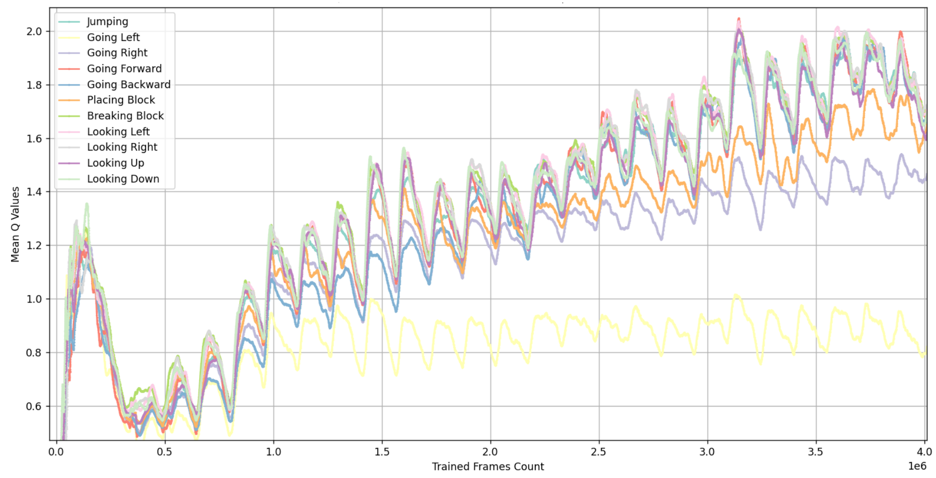
4. Conclusions and Future work

This project was, for some of us, the most interesting they had ever worked on. It gave us a lot of room for creativity and exploration, and we believe we were able to harness it to create a game of our own, a procedural generation and sandbox environment of our own, a deterministic agent of our own, a Deep Reinforcement Learning platform of our own, and even a hardware-maximizing multi-clients multi-servers, training setup, akin to what was seen in state-of-the-art RL papers less than a decade ago.

Still, this task, of learning to play a sandbox game in 3D just from player-camera pixel data, is an immensely difficult one. One that is known for possibly taking the most convoluted Deep Learning systems, and the longest, most intensive computations in all of this field of research. It opened our eye on so many aspects of training game AIs. From designing the environment, making it run enough at scale, and creating robust tools to speed up, secure and monitor the training procedure. There are so many new approaches that we found in the literature that we would like to try, gameplay elements we would like to implement, and environments we would like to experiment with. We feel confident doing so after this project.

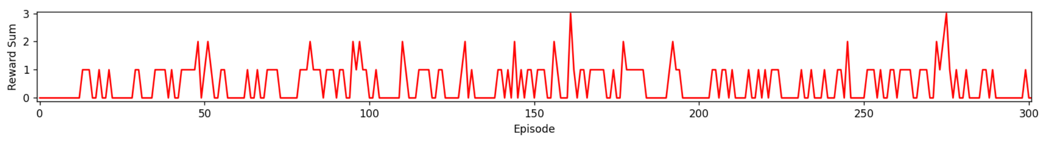


(a) From-scratch's Q-values for each action over first 3.5M trained frames

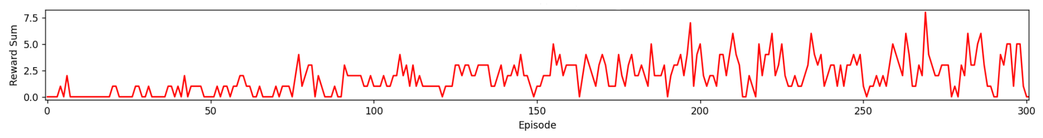


(b) World-aware's Q-values for each action over first 4M trained frames

Figure 14: 14a it is quickly learning to want to break blocks at the beginning, but after that it is just chaotic and doesn't reach very high. Either we are looking at a small zoomed-in pattern in what would be a 100M frames-long learning curve, or simply a model-free Q-learning approach is not giving enough signal to the model to find consistency and patterns. 14b while the World-aware model trains smoothly, oscillating at the rhythm of the episodes starting and finishing. Still, we see the drawback of doing exploration only via Noisy Linear layers. It is good for exploration, but once it outclassed some output for dubious reasons (like the going left action here), it cannot always get it back up. Also, it will clearly soon reach a higher, but not super impressive plateau.



(a) From-scratch max validation reward sum over 300 episodes



(b) World-aware max validation reward sum over 300 episodes

Figure 15: 15a is stuck on a plateau and shows no sign of learning, while 15b does, and even achieves to get 7 precious blocks in a run, which is not trivial at all, but this progression still hints an upcoming plateau.

Task	Assigned to
1.1 Creating initial flat world	Jens
1.2 Research ML Agents	Jan, Maciej
1.3 Research Deep Reinforcement Learning	Anicet
1.4 Writing report	Anicet
1.5 Research cooperation tools with Unity	Jan
1.6 Environment set-up	Everyone

Table A.1: Planned tasks

Appendix A. Sprint 1, Oct. 23rd, 2023

Appendix A.1. Planned tasks

Appendix A.2. Work Done

Creating the initial flat world, see Figure A.16 (1.1) was done by trying out different grid patterns and searching for fitting assets in Unity's asset store.

So far, everyone tried to get an initial understanding of the ML agents (1.2) that Unity offers by watching tutorials and trying to implement it by themselves. The focus of this sprint lies more on getting more experience with Unity, as none of us have worked with Unity before.

Appendix A.3. Work in progress

At this moment, we should focus on finding ways to efficiently train the ML agents (in parallel) and start to implement the different components of

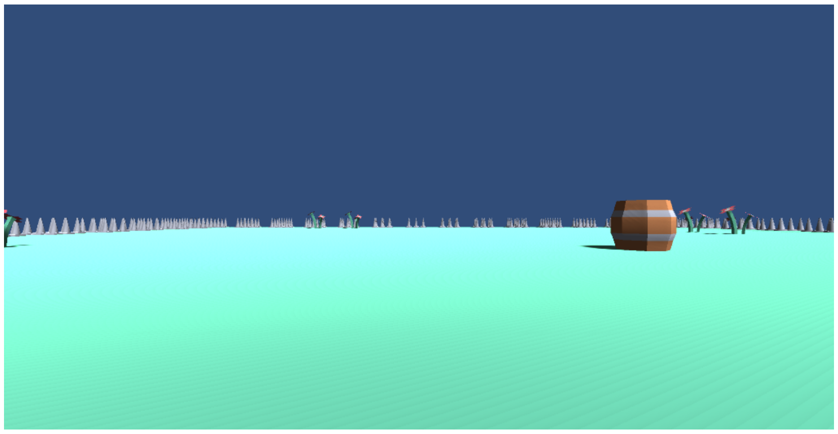


Figure A.16: View of what the initial flat world looks like

our game into Unity. This includes designing the different agents, procedural world generation and training the AI components of the game. Finally, attention should be paid to the general flow of the game.

Appendix B. Sprint 2, Nov. 20th, 2023

Appendix B.1. Planned tasks

Describe planned tasks and who is the responsible for each task

Task	Assigned to
2.1 Player movement	Jens
2.2 World generation	Maciej, Jens
2.3 Mesh & rendering	Jens, Anicet
2.4 Multiple worlds	Anicet
2.5 Programmatically controllable player & camera	Anicet
2.6 Communication with the model	Anicet
2.7 Enemy AI	Jan
2.8 Setup collaborative tools and sync	Jan

Table B.2: Planned tasks

Appendix B.2. Work Done

Having player movement (2.1) is a crucial aspect of every game. At first, the plan was to only implement movement and leave the rotation unimplemented to make it easier for the AI. The rotation would however be needed to place and break blocks in a later stage. The main difficulty was the fact that the world is randomly generated, which made the detection of the ground harder.

For the world generation (2.2), which is a crucial part before any of the other part can be developed, was finished in this Sprint.

Once we had generated world data, we needed to render them (2.3). The main difficulty was figuring out this whole process. Fortunately, it was documented online, as many other game creation enthusiast are making minecraft clones in Unity.

Having multiple worlds (2.4) required to further refactor the project as to allow generating worlds multiple times with different seeds at different location, see Figure B.17. It proved that the rendering approach was solid, by scaling smoothly to 100 concurrent $20m^3$ worlds.

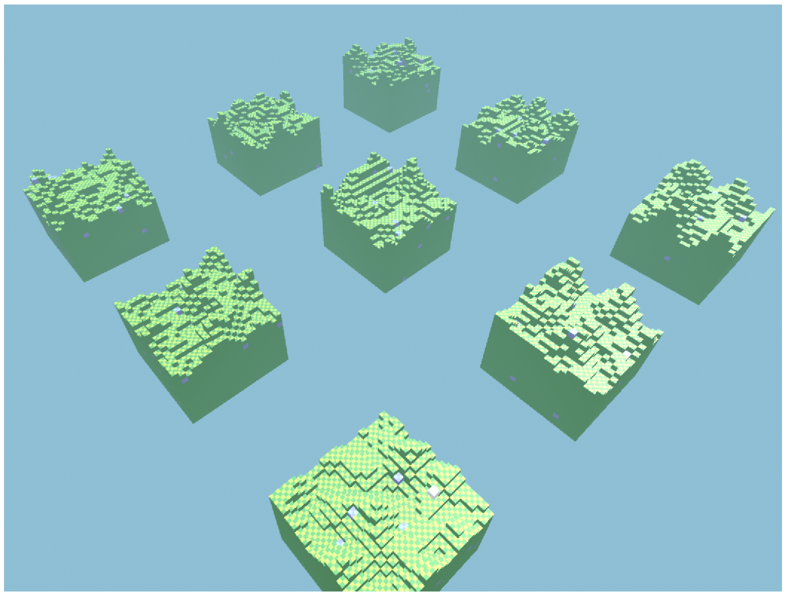


Figure B.17: Example of the generation of multiple parallel worlds

Furthermore, on each world, a dedicated player (2.5) is generated to be controller via a training procedure instead of the human inputs. For this to work smoothly, communication (2.6) was set up, as described in Section 3.5.3

On each world, we generate programmatically a dedicated player (2.5), which script was modified as to be controllable with programmatic inputs rather than user input. For now, we also spawn a dedicated static camera for each world which renders a slightly top-down view of it at a low resolution, as to serve as an input for the player's model. A centralized system, called the Conductor, was implemented to run all the players and cameras in sync at a constant rate and receive incoming instructions for each of them, as the goal is simulation, not real-time play.

The enemy agent, called the Creeper (2.7), is being investigated too. Progress was made, but difficulties still arise on the pathfinding, as Unity's built-in pathfinding is difficult to adapt to the cube-based worlds where navigating requires jumping over blocks and gaps. A custom algorithm using a modified A* implementation online can always be considered later if Unity's system ends up not being adaptable.

Finally, early during the sprint, a solution using Git was found to sync everyone's work (2.8). It is now used by everyone effectively.

Appendix B.3. Work in progress

The following tasks are still a work in progress:

- Implementing a rigorous training/testing setup
- Implementing the enemy AI and player kill mechanic
- Implementing world edition and block updates rendering
- Implementing player aim-based block breaking/placing, along with the inventory counters
- Implementing optional UI elements, menus, and game flow

The following tasks are under research but are not being implemented yet:

- Training a basic AI using Deep Reinforcement Learning that learns to reach the highest peak of its world
- Training an advanced AI that learns to gather precious block without dying

Appendix C. Sprint 3, Dec. 18th, 2023

Appendix C.1. Planned tasks

Describe planned tasks and who is the responsible for each task

Task	Assigned to
3.1 Creeper AI	Jan, Jens
3.2 UI, start screen	Maciej
3.3 Placing & breaking blocks	Jens, Anicet
3.4 Background music	Maciej
3.5 Training AI agent	Anicet

Table C.3: Planned tasks

Appendix C.2. Work Done

In the sprint, the AI of the Creeper (3.1) was finalized. On top of that, it was tested to ensure a smooth gameplay.

During this sprint, we also managed to implement a start screen (3.2) that allows us to set up our basic gameplay. We are able to adjust the number of creepers, the size of the world and the frequency of diamond blocks generation.

Also we managed to add an UI to the game (3.2), where we can preview the remaining time, the number of blocks collected. While holding down the F3 key, we are also able to see additional parameters such as the current position of the player (X, Y, Z coordinates), the position of the creepers and the distance of the creepers from the player.

The placing and breaking of blocks (3.3) required a complete restructure of the code by being able to access the meshes structure for a given coordinate. This way, it became possible to change the specific voxel of a given 3D coordinate.

Progress was made on (3.4), as visible in sections 3.3.2, 3.3.3 and 3.5.3. We may also note that many more models were trained with various setups and environments, as to progressively reach the current state of complexity. So far, no satisfying model has been trained, but promising hypothesis were formulated, and mitigation options, often requiring an increase in complexity, allowed us to observe progress in model quality and stability. This gives us hope of successfully training an at least smarter than random AI, in the desired, quite challenging settings.

Appendix C.3. Work in progress

AI training and UI wiring are still a work in progress.

References