Dirt House

Craig Knoblauch

December 7, 2018

Abstract

In this report, I'll discuss the process of training a robot to build a dirt house in Minecraft, using Deep Q-Learning. In the introduction, I'll be giving an overview of the project. I'll discuss the scope of the project, the technologies used, justifications for those technologies, and I'll wrap up with a brief about terminology used in this project. The next section will focus on the Minecraft side of the project. An entire API was developed for interacting with Minecraft. This section explains the components of that API. The second section details the AI implementation. I'll discuss how the actions of the agent are realized, and how the agent receives its rewards. In the final section, I'll discuss the results of the project, as wells as plans for future research.

1 Introduction

The goal of this project was to design, develop, and train an AI agent that could interact with the environment shown in figure 1. The goal for the agent would be to take dirt blocks from the pile, and build the house shown in figure 2.

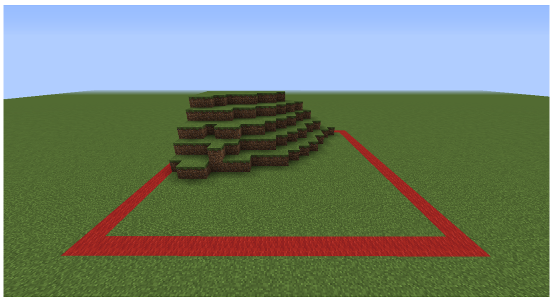


Figure 1: A blank environment

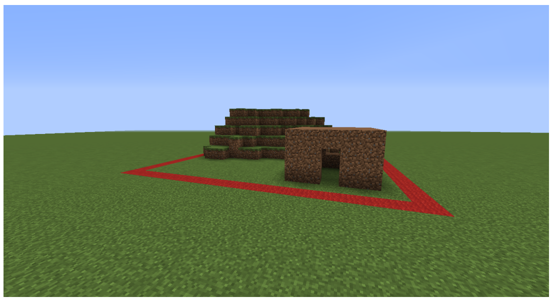


Figure 2: The dirt house the agent would build

To accomplish this goal, two programs were needed. One program, worked within the game to interact with the environment. On the outside of the game, but on the same machine, a second program gave commands to the first and used the responses to train the AI agent. The programs communicated with each other over a TCP link on localhost (127.0.0.1).

Inside the game, I used a robot provided by the OpenComputers mod. This mod adds computers and robots to the game, that are programmable with Lua 5.3[1]. The robot API provided by OpenComputers provides several useful functions for controlling the robot and interacting with the game. Since I would have a robot operating in a specific environment, with specific goals in mind, I decided to build my own API around the existing robot API. This API will be described at length in section 2.

Outside the game, on the same machine, a python program operates an AI agent. This agent uses Deep Q-Learning and a Deep Q-Network (DQN) to train. The program directs the in-game robot through a series of episodes to learn, through trial an error, how to build the house. Shown in figure 3, are what these episodes appear as, in-game. The agent, it's design, development, and operation will be discussed thoroughly in section 3.

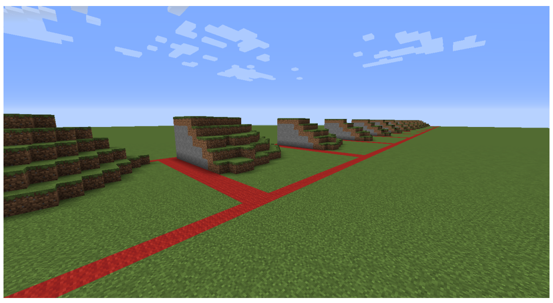


Figure 3: Episodes shown in game

The OpenComputers mod was heavily inspired by an earlier mod, ComputerCraft. Personally, I was introduced to ComputerCraft first, but decided to use OpenComputers for this project because of its active community, and richer set of features. In OpenComputers, robots were referred to as turtles. Keeping with the tradition, I decided to name my programs after an infamous pair of turtles, Squirt and Crush[2]. From this point onward in this report, "Squirt" will refer to program components in game, while "Crush" will refer to the Python program operating out of game, and training the AI agent. There are some mentions to the "East Australian Current (EAC)." This is a python class responsible for interpreting information about Squirt's environment.

2 Squirt - The in-game program

Squirt is the physical entity interacting with the Minecraft world. This section will describe both how I developed an API to control Squirt, and the components of OpenComputers that enable Squirt to function.

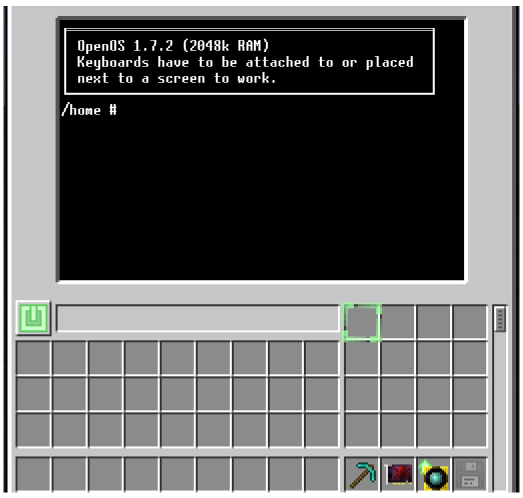
Squirt is commanded by Crush to perform one of 10 actions in each episode. These actions include:

1. Go forward
2. Go back
3. Go up
4. Go down
5. Turn left
6. Turn right
7. Turn around
8. Pick up block
9. Place dirt block
10. Place cobblestone block

Even though actions 1 through 7 seem trivial, I still developed custom implementations for them in my API. By doing this, I achieved two things. One, if I ever had a problem with Squirt moving, I could trace the problem back to one place. Two, if Squirt ever needed to do anything additional in one of these movements, I would only have to modify one section of code. This decision turned out to be a fantastic idea as I developed Squirt's API. Before we talk about the modules I developed to meet these functional needs, I'd like to take section to discuss how I developed Squirt.

2.1 Development

Since I could only interact with Squirt in-game, his development was arduous to say the least. Shown below in figure 4, is everything that I see in-game when I operate Squirt.



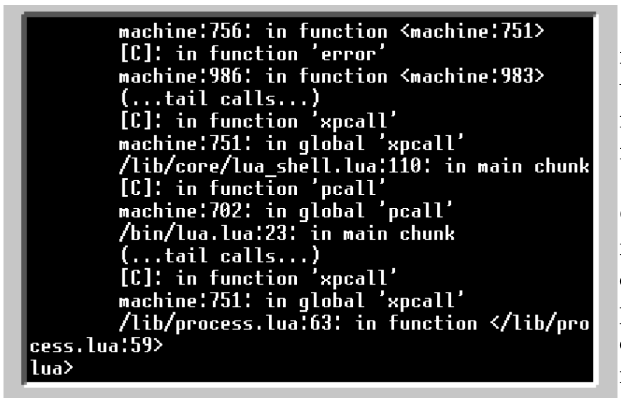
```
OpenOS 1.7.2 (2048k RAM)
Keyboards have to be attached to or placed
next to a screen to work.
/home #
```

Figure 4: Squirt's terminal, and inventory

My inventory is is shown in the bottom left. Squirt's inventory and equipment are shown in the bottom right. The top of the image shows the terminal in which I can run lua scripts, configure files in Squirt's file system, and "debug" code (Figure 5 shows just how useful Squirt's errors are. Figure 5 is the same output I saw for every error).

2.2 Functional Modules

As described at the beginning of section 2, I developed several modules on top of the OpenComputers API. I'll now take a few sections to discuss each module. Its development, testing, and challenges will be discussed.



```
        machine:756: in function <machine:751>
        [C]: in function 'error'
        machine:986: in function <machine:983>
        (...tail calls...)
        [C]: in function 'xpcall'
        machine:751: in global 'xpcall'
        /lib/core/lua_shell.lua:110: in main chunk
        [C]: in function 'pcall'
        machine:702: in global 'pcall'
        /bin/lua.lua:23: in main chunk
        (...tail calls...)
        [C]: in function 'xpcall'
        machine:751: in global 'xpcall'
        /lib/process.lua:63: in function </lib/pro
cess.lua:59>
lua>
```

Figure 5: Very helpful error output

Thankfully, I didn't have to do all of my programming in Squirt's terminal. OpenComputers provides their own package manager, OPPM. With this package manager, I was able to develop outside of the game, push my code to my repository, and update Squirt with OPPM.

2.2.1 sq-navigation

This module is a core piece of functionality for Squirt. Fundamentally, this module is responsible for keeping track of Squirt's position, both in the world and in the episode, and Squirt's direction. That distinction, between the world and the episode is very important. Crush needs information about Squirt's actions in an episode, but the base robot API only knows information about it's place in the world. To eliminate this disparity, sq-navigation initializes 4 global variables on startup.

```
FACING = "east"
SQ_INIT_X = nav.sqGetX()
SQ_INIT_Y = nav.sqGetY()
SQ_INIT_Z = nav.sqGetZ()
```

The sq-navigation module then uses it's own internal functions to determine where Squirt is in an episode. Below is some simple source code showing how Squirt's ”episodic position” is determined (I'm sorry for the word wrapping).

```
--[[ Return Squirt's Z position
   based on the episode count ]]
function nav.sqGetEpisodicZ(episode
   )
    local epZ = nil

    -- 18 because each starting
       point is 18 blocks away
    epZ = nav.sqGetZ() - SQ_INIT_Z
       - (18 * episode)

    return epZ
end
```

In some cases, such as picking up or placing a block, Crush must know the coordinate which was acted upon. In these cases, the sq-navigation module must determine the episodic coordinate being acted upon.

Originally, I had planned to use OpenComputer's navigation upgrade, an attachable item for OpenComputer robots, to fulfill this need. I decided though, that I might as well build wrapper functions around the navigation upgrade's get coordinate function calls. Much later in development, as I was finishing Squirt, I found that if a player left the chunk Squirt was operating in, Squirt would cease operation. To fix this, Squirt had to be equipped with the Chunkloader upgrade (you can see the Chunkloader upgrade in figure 6). Unfortunately, OpenComputer robots only have the capability to house one upgrade. This meant I had to ditch the navigation upgrade. Fortunately, all I had to do was modify a few lines of code to make the switch seamless.

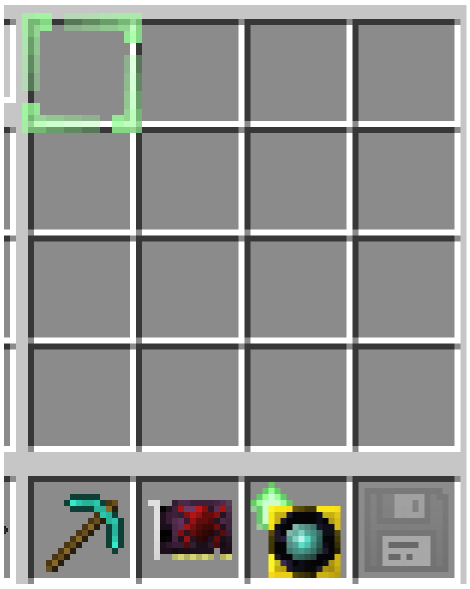


Figure 6: Squirt's inventory. From left to right: equipped tool, debug card, Chunkloader upgrade

2.2.2 sq-swim

The sq-swim module handles all cases where Squirt must move within his environment. In a broad stroke, these functions include moving forward, backward, up and down. Turning left, right, and around. And going to the next episode. The OpenComputers robot API provides simple implementations for all requirements except going to episodes. The API implementations were not enough, however.

In my initial experiments with Squirt using the robot API, I found that the movement commands were non-blocking. If I commanded Squirt to move and then immediately recorded his position, Squirt report he had not changed positions! Therefore, for all movement functions I had to implement a blocking while loop. This loop would only exit when Squirt's position changed. You may have anticipated that I'd like the movement functions to fail if Squirt tries to wonder out of bounds of the episode. While this is true, I wanted to stop this command higher up in the API (you can read about this in section 2.2.5). Working with this design meant that the decision to move or not happened in one place, rather than every movement function.

I also decided to wrap the turning robot API calls. While at the time of coding this module my wrappers did nothing besides call the robot API, later in development I needed this functionality to change. When that time came, I added the ability for the turning wrapper functions to call an update facing function from the sq-navigation module. Again, since the code was only changed in one place, the rest of the code base was unaffected by the change.

2.2.3 sq-act

The sq-act module is responsible for managing Squirt's interaction with the environment. Coming in at 733 lines, the sq-act module is the most expansive module I developed. Most of the functions support the primary features: picking up a block, and placing a certain kind of block. To do both of these actions, Squirt's inventory has to be managed in specific way. Refer back to figure 6. Squirt's inventory slots are ordered, starting from 1, from the top, left to right. Slots 1 through 4 are reserved for dirt, while slots 5 through 8 are reserved for cobblestone. If Squirt's inventory is full, the picked up block will roll over into the next available slot. The pick up block and place block functions must go through the logic of determining where in the inventory to select to either put a block in, or draw a block from. It's worthing noting here, that the identity of the block is determined utilizing the debug card, shown in figure 6.

You may ask, why not allow Crush to control Squirt's inventory. While I considered this in the beginning, this would meaning expanding an already large action space.

In my initial tests, I noticed that Squirt would break a block, but not add the block to his inventory. I determined that Squirt had to be equipped with a tool (shown in figure 6). The only tool capable of breaking both stone and dirt is a pickaxe. This is not an ideal tool. The pickaxe takes a long time to break dirt, thus increasing the amount of time Squirt would spend in episodes.

2.2.4 sq-comms

The sq-comms module is responsible for establishing communications with Crush. Crush creates a server on localhost, and opens port 65432. Squirt connects to the port to read and write. The OpenComputers internet API does a sufficient job at providing this functionality. I discovered a few things during testing that I decided to address with my own modifications.

For starters, the read function requires a number of bytes to be read. I decided to hardcode this to 3. This enforces a standard that Crush must meet when writing to Squirt. sq-comms also maintains a persistent handle on the connection. When connect is called for the first time, this handle is set, but it's only the responsibility of the module to interact with it. This removes responsibility from the main Squirt program.

I also decided to custom define the write function. I wanted Squirt to return a standard string for easy parsing. I therefore had sqWrite accept relevant arguments, and write these components to the server:

1. step
2. episode
3. x
4. y
5. z
6. direction facing
7. action code
8. return code

2.2.5 sq-exe

This module is responsible for translating commands from Crush into actions within the environment. The implementation for performing actions is very straightforward. This module simply collects calls to the other modules in one place. The one specialized operation this module has is controlling the bounds of Squirt's operations.

Back in section 2.2.2, I discussed how the control of bounds is left to a module higher level than sq-swim. sq-exe controls these bounds. Before any action is executed, sq-exe determines if the action would be taking Squirt out of bounds, or affecting a block out of bounds. If either of these occur, the action will not be executed, and sq-exe will return -10. Not allowing Squirt to operate in these boundaries achieves an Atari like game. In Atari games, a player can try to move the player offscreen, but the player will not move. In the same way, Crush can command Squirt out of bounds, but Squirt will not comply.

Clearly, getting this piece of functionality right was crucial to Squirt's success in an episode. I wasn't leaving anything to chance, and I decided to build a cage to test the boundaries in. Shown in figure 7, this cage matches the boundaries of an episode. It took several hours to design and implement a testing script. Had I not though, and a few nefarious bugs would've gone unnoticed, and Squirt could have easily wandered outside the boundaries during training.

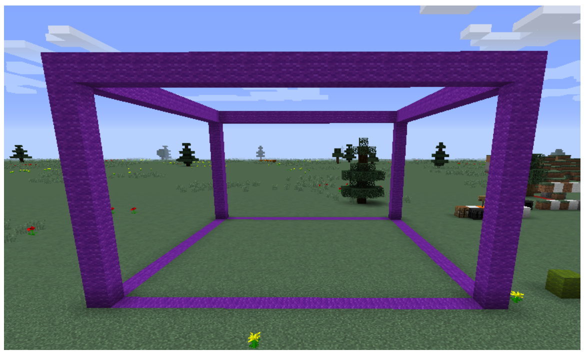


Figure 7: 16x16x9 area in which I tested Squirt's boundary recognition

2.3 squirt.lua - The main program

With all the functional modules implemented, I could create the main program to run commands from Crush. The design was to have Crush save the model every episode. Because of this, I decided to let Squirt run in an infinite loop.

3 Out of game - EAC, Crush, and the DQN

In this section, I'll be discussing the design, development, and test of these 3 major components. I'll start with the EAC. This classes responsibility is to translate the information from Squirt for Crush. Next I'll discuss Crush. Crush is largely responsible for interacting with the DQN. Crush interprets information from the EAC to assign rewards to the DQN. Crush will also interpret action selections from the DQN to something Squirt can understand. Finally, the DQN agent is responsible for training the house building model. The DQN agent featured in this project, was reused from another project of mine: CraigKnoblauch/Raymond[3]. This agent, was also highly influenced by a tutorial cited here[4].

3.1 The EAC

If you remember Finding Nemo, the East Australian Current is the medium through which Squirt and Crush traveled[2]. So too, in this project, the EAC class is responsible for interpreting messages from Squirt to Crush. and vice versa. To accomplish this bidirectional communication, the EAC holds 2 crucial dictionaries.

The first dictionary of concern holds action codes. These bytes objects, shown below, refer to the actions listed in section 2.

```
'forward': b'001',
'back': b'002',
'up': b'003',
'down': b'004',
'turn␣left': b'005',
'turn␣right': b'006',
'turn␣around': b'007',
'pick␣up␣block': b'008',
'place␣dirt␣block': b'009',
'place␣cobblestone␣block': b'010'
```

Recall from section 2.2.4, I had hard-coded the number of bytes Squirt reads to 3. This is why the bytes objects shown each have 3 characters. The action codes listed here, are the same as those listed in sq-exe. While this is a simplistic implementation, ideally I'd like these codes defined in the same place. If this were the case, if the codes ever changed, I'd only have to edit one place.

The second dictionary holds two lists of coordinates. Both these lists are episodic coordinates that belong to the house. The only difference between the lists are whether the coordinates are filled with dirt or air. I've shown a small snippet of this dictionary below.

```
'dirt': [ (4, 2, 11),
          (4, 2, 12),
          (4, 2, 13),
          (3, 2, 13),
                .
                .
                .
'air': [ (1, 0, 12),
         (1, 1, 12),
         (4, 0, 11),
         (4, 1, 11),
         (4, 0, 12),
         (4, 1, 12),
         (4, 0, 13),
         (4, 1, 13),
         (3, 0, 11),
         (3, 1, 11),
         (3, 0, 12),
         (3, 1, 12),
         (2, 0, 11),
         (2, 1, 11),
               .
               .
               .
```

The EAC uses these coordinates to help Crush determine if one of Squirt's actions has affected the target space. For example, if Squirt places a dirt block while in position (3, 2, 11) facing east (+X axis), it's up to the EAC to determine that the affected position, (4, 2, 11), should indeed have dirt block placed in it. When asked, the EAC provides that information to Crush, which he uses to craft a reward.

It's also worth noting that the EAC class contains the definition for the host and port, but is not responsible for opening or sustaining the connection to Squirt.

3.2 Crush

To work seamlessly with the DQN agent from my previous project, I decided to have Crush inherit OpenAI's gym.Env[5]. This meant I could use known types for my action space and state space. Using gym.Env also meant I had a template for Crush. I only needed to define a few functions (step, reset, and render) to know Crush's implementation would be able to work with the DQN. In the next few sections, I'll discuss Crush's implementation of an action space, a state space, and the functions step, reset, and render.

3.2.1 Action Space

In section 2.2.5, I mentioned that I wanted the operation of Squirt to be Atari-like. In terms of an action space, this means discrete actions. OpenAI's gym gives an action space type for this goal. As shown in section 2, and again in section 3.1, there are a total of 10 actions to consider. Therefore the action space is created like so:

```
action_space = spaces.Discrete(10)
```

When the DQN generates actions to be taken, the action will be a number relative to this space. This number is translated by the EAC into 3 byte action code, that Squirt can understand.

3.3 State Space

This is where, unfortunately, the program starts to get a little hackhappy. OpenAI Gym requires the state space to be a single numpy array of numbers. Originally, I had planned this array contain tuples of coordinates, a boolean to represent if a block existed in that position, and a string with the name of the block in that position. So the state space, as I had planned, would look a little something like below.

```
[ (x1, y1, z1), block_exists, 'name
    ', (x2, y2.... ]
```

I quickly discovered that the APIs I was using for the deep learning agent, would not accept state spaces with sequences or strings. The names were an easy fix, just use an enumerated data type (fancy way of saying numbers for names because Python doesn't technically have enumerated data types). The coordinates on the other hand, I made indexes that related to the coordinates of the EAC's house dictionary (refer to section 3.1). I modified the EAC to support this, and the implementation, seemingly worked, but I'm not proud of this solution in the slightest. The solution is incredibly fickle. It requires me to scrutinize what the hard-coded number is in 3 sections of the code. A much better solution, would involve creating a coordinate class, and having the state space hold objects of that class. If objects are not allowed in these spaces. I could also create hashes for each object. These hashes would appear as numbers to the deep learning algorithm, but would be far easier to work with programmatically.

A final note on the state space. My implementation of the state space only accounts for the coordinates of the house. Squirts actions on the rest of the environment do not affect the state space. I don't think this was a good design decision. The deep learning agent operates both on rewards for actions, and affects on the state. In early training, the deep learning algorithm could be deprived of vital state information.

3.4 Crush.step

OpenAI Gym requires the step function to take in an action code, progress the agent through the action, determine a reward, determine if the agent is done, and return this information. Crush's implementation follows this outline, with a slight edit. Squirt operates outside of the step function. Therefore, I provide the outcome of Squirt's action to the step function. Inside the step function, Crush uses the action code, and the outcome to determine a reward (For a look at how rewards are assigned, refer to figure 8 in section 4.3). Crush determines that Squirt is done when the outcome includes a step count equal to 399.

As you can probably tell from the previous section, I'm not a fan of hacks. This implementation of the step function has a few of them. For starters, I shouldn't be modifying Gym's method signature. While it's not technically harming the programs in any way, it's bad practice when working with API's[6]. If my code does not follow an established convention, it will harm the codes flexibility to changing situations. Secondly, 'done' being directly dependent on step count is bad design. If I ever want to change the number of steps Squirt takes per episode, I have to remember to change this function, a crucial piece of this project. To solve both of these, I can move the transmission of Squirt's commands to inside the step function. This way, the step function behaves as it is meant to. Also, Crush can be in control of when Squirt ends an episode, thus eliminating the need for a hard-coded episode count in several places.

3.5 Crush.reset

This function resets the state space to it's default, where no coordinates of the house have been affected. Remember from section 3.3, the state space is hard-coded in a fickle way. That hard-coding is repeated in this function. Ideally, the setting of the state space should be handled by one function.

It's also worth noting, I programmed the ability to record the state at the end of each episode. I do this in the main Python program, but I'd rather have this happen inside this function.

3.6 Crush.render

This is a very simple function. Recall from section 2.2.4 that Squirt transmits specific information. This information is parsed by the EAC, and displayed to the operator using this function. Also in this function, each message from Squirt is broken into its components and written to an action record. In section 5, I'll discuss the insights this record gives into the training.

4 The Deep Q-Learning Agent

Out of all the possible learning approaches, why use deep q-learning? Deep Q-learning provides incredible solutions for problems with finite action spaces, and countably infinite state spaces. This description matches this project perfectly. Also, there are enumerable examples of deep q-learning agent implementations to follow. As this was a learning experience, I figured this was a blessing. Finally, API's such as OpenAI Gym, TensorFlow, and Keras make the process of implementing these architectures very simple (though not so simple to troubleshoot).

As briefly mentioned earlier in this report, I reused a deep learning agent from a past project of mine[3]. I believe this was a key mistake. The problem presented in that project, had both far smaller state and action spaces. There was less time the agent needed to explore, and much less steps the agent had to take to reach the end of an episode. In this section I'll discuss the reason for a few key components, and also how I would change them for improved performance (I'd consider this to be a little foreshadowing for section 5).

4.1 Memories

This is an interesting component of deep q-learning. It's the concept of saving actions, and the result of the state space because of them. While this is effective in other projects, for it to be effective here, Squirt's actions each must have affected the state space. Recall from section 3.3, the entirety of the episode is not in the state space. I believe this detail makes the use of memories ineffective. This is another reason why the state space implementation should be rethought.

4.2 Exploration vs. Exploitation

I believe this operation is one of the most important for the success of this agent. Since Squirt starts away from the house area, he must do a lot of exploring before he can begin understanding the state space. Also, there are dynamics involved with Squirt's exploration. Squirt must explore to the point where he understands the pile he must take blocks from, and the section of his episode he must add these blocks to.

4.3 Rewards

Commonly, in problems where an agent is moving about in a world, the reward structures will give small negative rewards for movement actions. The theory here, is that this will motivate the agent to exploit the fastest to the desired state. In my project I implemented this for movement. Common movement actions such as go forward, go back, turn right, etc. result in a small -1 reward. Larger actions, like picking up blocks, results in a higher reward. I wanted to reward picking up blocks in general, so regardless of the block type, it's a hefty positive reward; the reward is, however, much greater if the block is dirt. I had similar logic with placing blocks. If a dirt block was placed anywhere, it would result in a hefty reward; the reward was much larger, however, if the dirt block was placed in a correct position. If the dirt block was placed in an incorrect place, i.e. a coordinate reserved for air in the house, a hefty negative reward would be issued. If a cobblestone block was ever placed, Squirt would receive a massive negative reward. It's worth noting, as you'll see in section 5 figure 16, Squirt very rarely placed a cobblestone block. You can view a flowchart of this reward structure below, in figure 8.

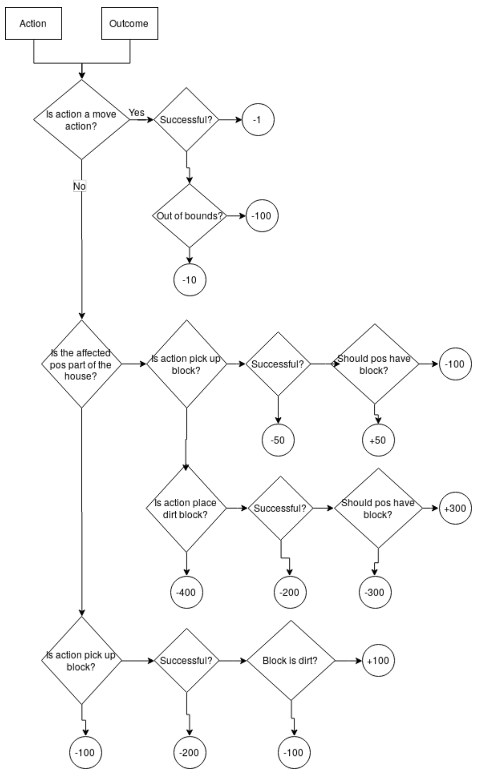


Figure 8: Reward structure

5 Results

So did Squirt learn to build a dirt house? Ultimately, no. Squirt did, however, make some semblance of progress. Over the course of 3.5 days (84 hours), Squirt completed 1,121 episodes in-game. After episode 500, Squirt consistently picked up dirt blocks from the same location and put them down again. Figures 9, 10, and 11 all show episodes where Squirt traveled to a nearby location of blocks, mined them, and placed a few down. Obviously, this is a far cry from making a dirt house, but it is some kind of learned, repeated behavior. In this section, I'll show how Squirt moved and interacted with his environment throughout the episodes. In each section, I'll briefly describe how I believe the Deep Learning Agent could be modified to achieve different results.

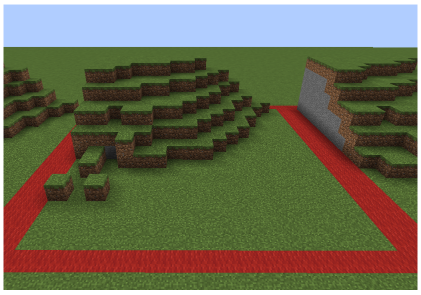


Figure 9: Example 1 of a visibly affected episode

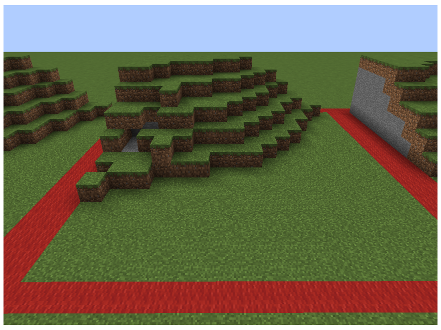


Figure 10: Example 2 of a visibly affected episode

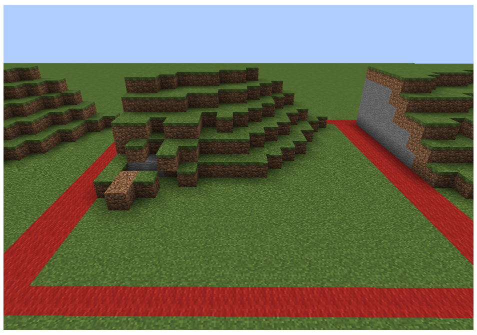


Figure 11: Example 3 of a visibly affected episode

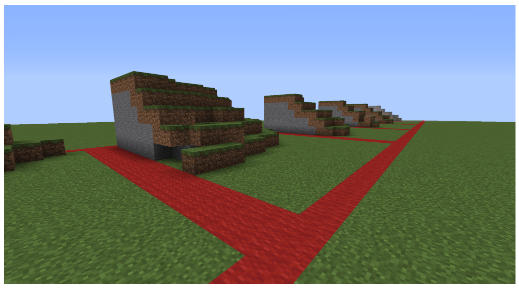


Figure 12: A sequence of visibly affected episodes

5.1 Squirt's movement in training

Figure 13 shows a heatmap of Squirt's movements in all episodes. You can see that Squirt almost never leaves the starting area (lower left hand corner).

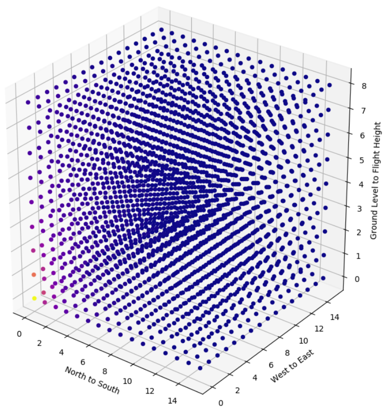


Figure 13: Squirt's most visited coordinates

I believe Squirt's attachment to the starting area, is a product of both the reward structure, and the aggressive exploitation. Squirt was disincentived from moving, and moved to exploiting this behavior too quickly. Squirt was never given the opportunity to explore outside of the starting area.

5.2 Squirt's actions during training

In this section I'll discuss how Squirt's actions changed throughout the training. This will be a mostly graphic discussion. First let's look at figures 14 and 15. These figures display the frequency of actions across all episodes.

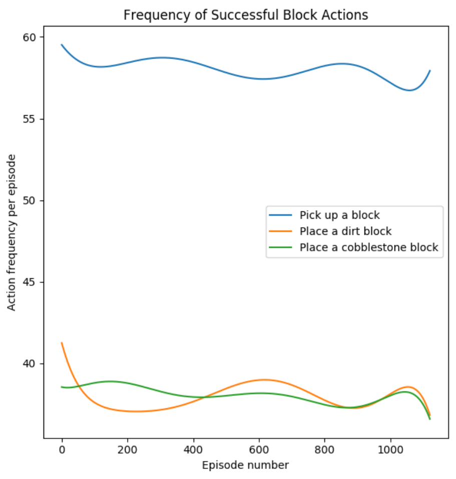


Figure 14: Frequency of block affecting actions across all episodes

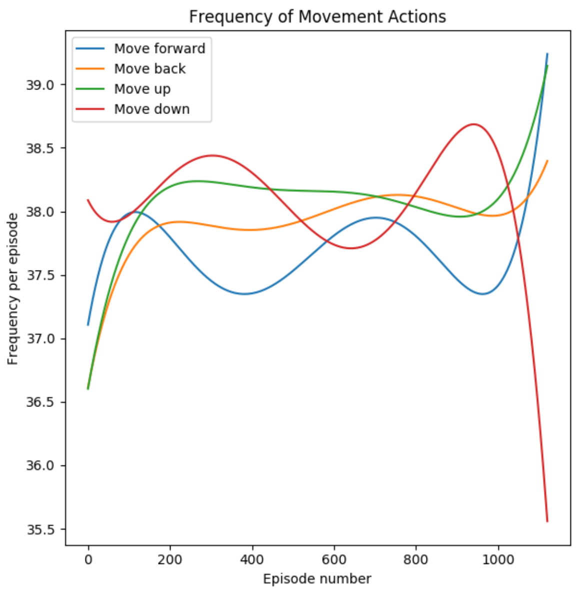


Figure 15: Frequency of move actions across all episodes

These results are concerning. Notice in figure 14 Squirt slowly stops trying to pick up blocks across the duration of the training. Even more concerning, Squirt immediately tries to stop placing dirt blocks. Both placements remain on a steady downward trend.

Figure 15 shows us the occurrence of move actions seems fairly stagnant, with exception given to the last 100 episodes. Squirt appears to suddenly disfavor moving down, and simultaneously favor moving in the other directions. Also of note, movement action frequency appears most heavily affected in the first 100 episodes. Perhaps this is because exploration is high.

Figures 14 and 15 showed all occurrences of these actions, but what about the frequency at which these actions are successful? Figures 16 and 17 show just that.

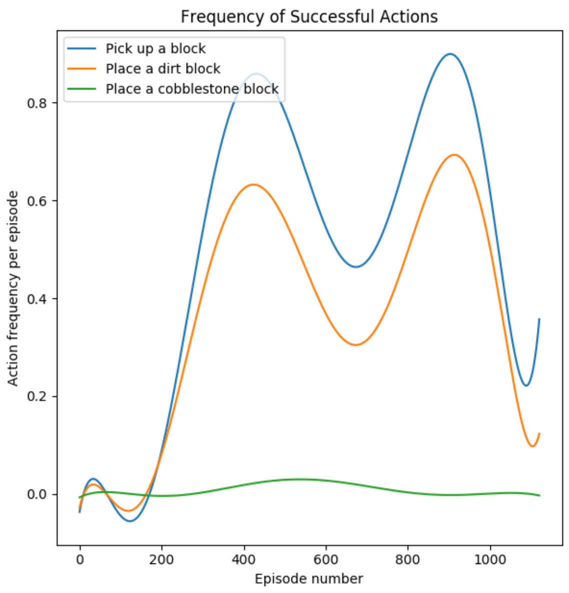


Figure 16: Frequency of successful block affecting actions across all episodes

On first glance of figure 16, it looks like the deep learning algorithm started to experience success around episode 200. Unfortunately, this is actually the result of fixing an operating error. Squirt needs a tool in order to pick up the block he breaks. Up until episode 200, Squirt didn't have a tool. It's unknown how much this affected training.

We see some successes in figure 17, with a slight upward trend in the success of move actions. This means, that Squirt learned to not travel out of bounds, and/or not travel into blocks.

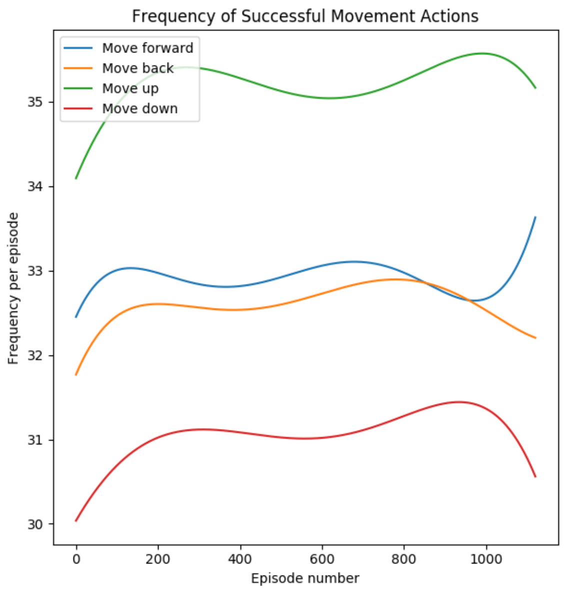


Figure 17: Frequency of successful move actions across all episodes

6 Closing Remarks

Was this project a resounding success? Was every goal met and the dirt house built perfectly? No. Was it a complete and utter waste of time? Absolutely not.

This project served as a fantastic education on just how difficult this technology can be. The ubiquitous tutorials can make it seem easy; with their tiny action spaces and comparatively less infinite state spaces.

So what are my next steps?

1. Refactor the state space implementation. Section 3.3 was wrought with criticism.
2. Refactor Crush.step (section 3.4).
3. Research different approaches to the exploration/exploitation problem. Squirt needs a lot of time to explore.
4. Address the criticisms laid out is section 2.2.
5. Find a solution for training that doesn't involve occupying one of my computers for a week. I'd like to explore how to get multiple instances of this training running on a server.

6. Create a standard set of plotting programs. The graphs and figures shown in this report gave me invaluable insight into Squirt's training, and they took hours to create. I'd like a script of plotting programs to run automatically whenever a training set completes.

I also believe in rolling the goal post back a bit. For the future of this project, I'd like to start Squirt with some amount of dirt blocks, a reduced amount of movement functions, and the ability to place and pick up blocks. With this criteria, the goal would be to build the house. If I can accomplish this, I believe I'd be primed to accomplish the main goal of this project.

This project has provided a foundation I can build on. I can use the tools and environment I've created here to truly learn how to use deep q-learning to solve this problem.

7 Source Code

If you would like to view the project, please follow this link: `https://github.com/CraigKnoblauch/dirt-house`. Just note, the README may be of little help if you're accessing this link shortly after the date listed on this paper.

References

[1] MightyPirates. (2018, November). OpenComputers. `https://github.com/MightyPirates/OpenComputers`

[2] Walters, G. (Producer), Stanton, A. (Director). (2003). Finding Nemo [Animated Film]. United States of America: Walt Disney Pictures and Pixar Animation Studios.

[3] Knoblauch, C. (2018, October). Raymond. `https://github.com/CraigKnoblauch/Raymond/tree/q`

[4] LiveLessons. (2018, September 21). Deep Q Learning Networks. Retrieved from `https://www.youtube.com/watch?v=OYhFoMySoVs&t=0s&list=WL&index=24`

[5] OpenAI. (2017, January 31). Gym. `https://gym.openai.com/docs/`

[6] Reddy, M. (2011). API design for C. Burlington, MA: Morgan Kaufmann.