ZedCraft:

FPGA-Based Minecraft Server Application

CPRE 488 Team S1-1

Simon Aguilar, Trevor Friedl, Ricky Smith, Justin Templeton

May 9th, 2024

1 - Project Overview

1.1 - Project Summary

CPRE 488 (Embedded Systems Design taught at Iowa State University) team S1-1 presents a new breath of fresh air into the life of FPGA computing, “ZedCraft.” The ZedCraft project was originally a concept to see the process of computing a Minecraft Server using Petalinux on an FPGA ZedBoard; however, this project expands upon different aspects of the 488 lab infrastructure by interfacing with the MP-2 (MP standing for “Machine Problem, and the number denoting the number of lab from class beginning at index ‘0’) FMC-IMAGEON camera module and the MP-3 nerf launcher turret. This project explores several practices of embedded system design, including cross-compilation, Linux programming, memory interfacing, open-source contribution, and many others. Throughout the project, our team aimed to try our best to document our process of setting up and using the server so that future students, instructors, and FPGA enthusiasts can learn and appreciate the work completed for this project. We hope that through this project, other embedded enthusiasts will learn and benefit from our work.

1.2 - Design Summary

Several components are interconnected throughout this project's duration. For the primary programming interface, our team used a Digilent ZedBoard, a Zynq 7000-based processor FPGA used within the curriculum of CPRE 488. More information on this class can be found here. Figure 1.0 Below details the high-level design of the system used for our project.

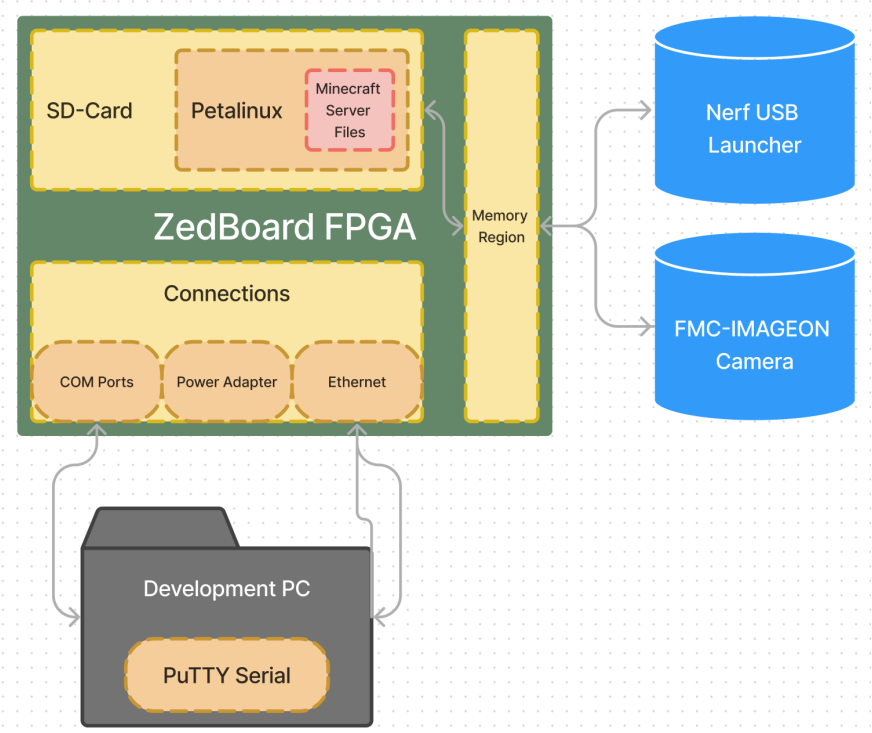


Figure 1.0 - ZedCraft high-level system design

The design details all of the primary connections responsible for running the work completed for this project. Using the ZedBoard, we use a Micro-USB connection to connect to the Nerf USB launcher, and we use the direct FMC interface connection on the ZedBoard to connect and communicate with the FMC-IMAGEON camera module. All external connections to the ZedBoard (besides power) transfer data between the different peripherals' memory spaces. The memory region on the ZedBoard is also partitioned and shared between the various peripherals, with the onboard memory limit not exceeding the physical 512MB limit from the board. Given this constraint, our design has been focused on targeting both memory-bound and performance-bound cases.

For storing the server data on the ZedBoard, we used an 8GB/16GB SD card to store critical server data, depending on the availability within the lab. The SD card writes and reads from data, including Minecraft server files, Minecraft world saves, and other essential functions to the ZedCraft server; overall, the SD card is primarily responsible for storing the configurations used for all of the server files and any other scripts we run to make starting the server much more manageable. One script to note is our *"startup.sh"* script, which we use to announce the memory configuration and startup of the Minecraft server. We use the below arguments to cast the memory overhead limits to the Minecraft server upon compilation:

```
"java -Xmx256M -Xms256M -jar server.jar"
```

The above command will execute the server.jar file, which starts up our Minecraft server with a 256MB memory size limit. We've found that 256MB is a valid amount of memory for the server, as allocating 512MB will result in a heap error. The server still requires a substantial memory allocation to hold different game operations, multiple players, and other computation overheads.

1.3 - Other Design Details/Requirements

It is also vital to describe other critical components of the project that were essential for getting things to work but do not require in-depth justification beyond its overall reasoning. These decisions include

Minecraft Client Version - Version 1.8.0

We chose to use a considerably outdated version of Minecraft because we believed that since the game has evolved, so have the memory and performance requirements. Going with a stable yet low-footprint build would be our best choice.

Minecraft Server "Flavor" - Spigot API

Minecraft allows for different "types" of servers to be implemented, all of which interface directly with the NBT (Minecraft packet structure) framework that enables data to be sent between Minecraft server and client. The Spigot API allows for an easy and efficient way for developers to program their custom logic in-game.

Java Build Version - Java 8

As stated in the Minecraft server documentation from the Mojang website, Java is not only required for both the server and client to be run, but both need at least Java 8 for Minecraft version 1.8 to run correctly. Matching server and client settings was required when installing Java onto our ZedBoard.

2 - Server Exploration

2.1 - Design Decision Justification For Java

One of the first significant decisions we needed to make was the possibility of getting the server to run on the ZedBoard. We had to think: "How can we even make this possible using engineering decisions?" At first, the challenge seemed quite scary to tackle since getting a Minecraft server to run on an FPGA has been unexplored territory, even getting Java to run on Petalinux. After researching, we came across a Yocto layer named "Meta-Java," an open-source meta-user layer catering to compiling dependencies for an ARM build of Java baked into Petalinux. Without any initial aim- we found that this would be the best-fit solution for getting a Minecraft server to compile on the ZedBoard. As much as we found that this would also allow for runtime optimizations, we had to consider the memory allocation for compiling this layer into the operating system instead of storing the JDK on the SD card *(spoiler: we found this to work as well)*.

2.2 - Yocto Meta-Java Layer

In this section, we want to address the Meta-Java layer further. Meta-Java is an OpenEmbedded and Yocto layer designed for compiling Java applications on Petalinux without using external dependencies, such as a local JDK saved on a storage device. This project has been around for several years and has not taken management over the last four to five years. The link to the GitHub repository can be found here.

Another significant issue that faced our project was that we needed to find a build of Meta-Java that we knew would interface correctly with the architecture of the ZedBoard. Fortunately, the developers of Meta-Java have a specific git branch responsible for targeting the "Zeus" architecture of the Zynq processing fabric. Given the availability of this architecture, we knew we had a good chance of getting Meta-Java to run on our board as long as we could guarantee that all resources could be fetched and appropriately compiled into our Petalinux build. Admittedly, Meta-Java was not the first implementation of Java on our ZedBoard for the project. Still, we would like to detail our process for getting the considerably outdated layer to work and be updated with our project.

The following content of this section is dedicated to exploring the steps required to get Meta-Java on our Petalinux build (more specifically, the *image.ub* file).

To first get Meta-Java registered on the board, we needed to get the Meta-Java repository cloned into our Petalinux project (additionally checking out the *"Zeus"* branch). We found that placing the repository in the *"project-spec/"* directory was best.

Afterwards, we needed to get the Meta-Java user layer registered within the project configuration. This required three main steps, all of which we will go over to replicate our work. First, we needed to type in the following command(s)

```
root:~# source /remote/petalinux/settings.sh
root:~# cd ~(project-dir)/avnet-digilent-zedboard-2020.1/
root:~# petalinux-config --get-hw-description ../
```

The three commands above are responsible for configuring Petalinux on the development virtual machine we used, and for getting us to open up the configuration screen for our Petalinux build. In Figure 2.0 below, we show the one argument needed to be added into the *"User Layer"* settings:

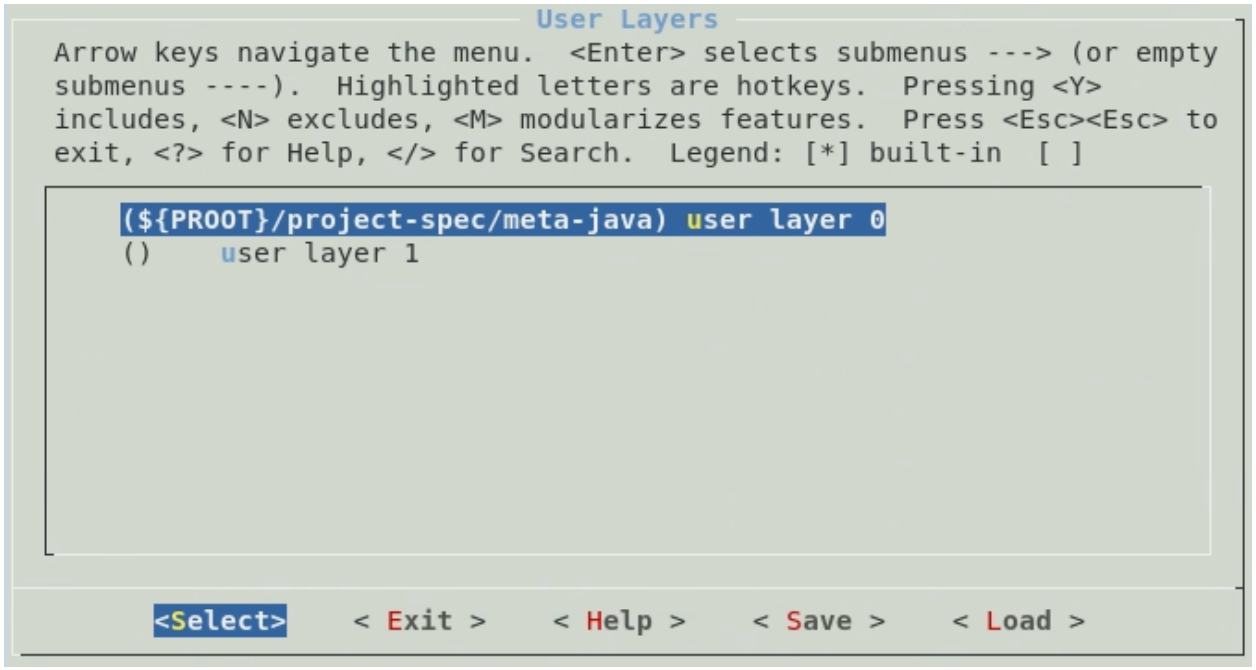


Figure 2.0 - Petalinux path configuration for Meta-Java user layer

Afterwards, we also needed to navigate over to our layers. A few changes were required so that we could tell our project to append the Meta-Java layer process, as we needed to specify Meta-Java as a layer to be rendered and to have its dependencies installed and included in the image.ub file. This was done by adding Meta-Java into the local.conf and bblayers.conf files. These edits are located under the petalinux main folder in ./build/conf/

local.conf

build > conf > local.conf

```
WITHIN_EXT_SDK = "1"


USE_XSCT_TARBALL = "0"

# Extra settings from environment:
MACHINE = "zynq-generic"

require conf/locked-sigs.inc
require conf/unlocked-sigs.inc
include conf/plnxtool.conf
include conf/petalinuxbsp.conf

# Added for Meta-Java configuration, steps found at https://git.yoctoproject.org/meta-java/about/
PREFERRED_PROVIDER_virtual/java-initial-native = "cacao-initial-native"
PREFERRED_PROVIDER_virtual/java-native = "jamvm-native"
PREFERRED_PROVIDER_virtual/javac-native = "ecj-bootstrap-native"

# Addition of the image layer install
IMAGE_INSTALL_append = " openjdk-8"

```

Figure 2.1 - Local.conf

bblayers.conf

build > conf > bblayers.conf

```
# WARNING: this configuration has been automatically generated and in
# most cases should not be edited. If you need more flexibility than
# this configuration provides, it is strongly suggested that you set
# up a proper instance of the full build system and use that instead.

LCONF_VERSION = "7"

BBPATH = "${TOPDIR}"
SDKBASEMETAPATH = "/home/tjfriedl/Desktop/final-project-petalinux/avnet-digilent-zedboard-2020.1/components/yocto"
BBLAYERS := " \
  ${SDKBASEMETAPATH}/layers/core/meta \
  ${SDKBASEMETAPATH}/layers/core/meta-poky \
  ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-perl \
  ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-python \
  ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-filesystems \
  ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-gnome \
  ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-multimedia \
  ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-networking \
  ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-webserver \
  ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-xfce \
  ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-initramfs \
  ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-oe \
  ${SDKBASEMETAPATH}/layers/meta-clang \
  ${SDKBASEMETAPATH}/layers/meta-browser \
  ${SDKBASEMETAPATH}/layers/meta-qt5 \
  ${SDKBASEMETAPATH}/layers/meta-xilinx/meta-xilinx-bsp \
  ${SDKBASEMETAPATH}/layers/meta-xilinx/meta-xilinx-pynq \
  ${SDKBASEMETAPATH}/layers/meta-xilinx/meta-xilinx-contrib \
  ${SDKBASEMETAPATH}/layers/meta-xilinx-tools \
  ${SDKBASEMETAPATH}/layers/meta-petalinux \
  ${SDKBASEMETAPATH}/layers/meta-virtualization \
  ${SDKBASEMETAPATH}/layers/meta-openamp \
  ${SDKBASEMETAPATH}/layers/meta-jupyter \
  ${SDKBASEMETAPATH}/layers/meta-vitis-ai \
  /home/tjfriedl/Desktop/final-project-petalinux/avnet-digilent-zedboard-2020.1/project-spec/meta-user \
  /home/tjfriedl/Desktop/final-project-petalinux/avnet-digilent-zedboard-2020.1/components/yocto/workspace \
  "

# Addition of the meta-java layer
BBLAYERS += "/home/tjfriedl/Desktop/final-project-petalinux/avnet-digilent-zedboard-2020.1/project-spec/meta-java"
```

Figure 2.2 - bblayers.conf

2.3 - Getting the Minecraft Server On-Board

The first implementation of Java that worked on the ZedBoard was through an ARM32 package of JDK v1.8.0_202 provided by the Oracle website. To run Java, we needed a **gcc compiler** to run and compile C/C++ on the ZedBoard, which was not enabled by default. We needed to navigate our Petalinux **rootfs** settings and include the "*packagegroup-petalinux-build essentials*" within the build. After doing this, we could get gcc to compile correctly, as seen below in Figure 2.3.

```
root@avnet-digilent-zedboard-2020_1:~# gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/libexec/gcc/arm-xilinx-linux-gnueabi/9.2.0/lto-wrapper
```

Figure 2.3 - GCC Compile message through PuTTY Serial

After confirming the successful operation of the C compilation, we then moved on to getting Java to run on the ZedBoard. Since there was little documentation available online about integrating Java into a Yocto environment, we knew this task would be possible; however *rather tricky*. After we were able to find a suitable JDK build for our project, we installed the contents of the JDK onto the SD card. The next step was registering the Java compile binaries added as "global variables" to compile Java throughout the Petalinux environment. After spending some time researching, we found that the following two commands enabled us to do so (ran from the root directory):

```
root:~# export JAVA_HOME=/../../mnt/sd-mmcblk0p1/jdk1.8.0_202/jre/
root:~# export PATH=$JAVA_HOME/bin/:$PATH
```

After both commands were run, we could finally see our first successful note telling us that Java had run on the ZedBoard through the serial port, as shown in Figure 2.4ls .

```
root@avnet-digilent-zedboard-2020_1:~# java -version
openjdk version "1.8.0_242-internal"
OpenJDK Runtime Environment (build 1.8.0_242-internal-ga)
OpenJDK Client VM (build 25.242-bga, mixed mode)
root@avnet-digilent-zedboard-2020_1:~#
```

Figure 2.4 - Java message through PuTTY serial

When getting Meta-Java to run on the board, we can completely ignore the above commands and simply type: "`java -version`" in the shell, and we should be able to see Java prompt us once again- denoting a successful build and install.

Now, we can move on to the contents of running the Minecraft Server. For this to work, we must establish an ethernet connection from the board into the network we want to connect to from the client. After connecting the ZedBoard via ethernet to a network, type: "`ifconfig`" and obtain the

IPv4 address listed in the terminal. **This is the address for your Minecraft server connection from the client.** The only difference is that you will append port 25565 to the IPv4 in Minecraft. Once all of the server contents have been saved onto the SD card, you should have a *"server/"* directory to access the server files (this should be copied from the cloned ZedCraft repo). From here, you should have everything you need to run the server.

2.4 - Server Optimizations

3 - Bash Script Exploration

3.1 - Server.sh

Server.sh is a bash file that starts the server as well as creating a FIFO socket that has its contents piped into the minecraft server. This allows for the input side of the server and petalinux communication.

```
1   #!/bin/bash
2   #Make sure test.sock is gone
3   rm /tmp/test.sock
4   #Make a new First-in First-out test.sock
5   mkfifo /tmp/test.sock
6   while true; do
7       cat /tmp/test.sock
8   done | java -Xmx400M -Xms400M -jar server.jar
9   # Cat anything in the socket and put it into the server.jar executable
10  #This will also start the server with 400Megabytes of RAM
```

Figure 3.1 - Server.sh

3.2 - Backdoor.sh

The backdoor is our method of getting data out of the minecraft server by tailing the currently active log file (named latest.log) for things that the users are doing. This included user deaths, messages, and many other things that can be used. We have three modes of use for the backdoor. First, turret control mode can be used by beginning a chat with ">>!". Normal usage of this mode would be ">>!10", which sends the byte \x10 to our turret, which is a USB Block Character Device. This can be used to control the turret from inside the game. Second, we have a mode that runs a command directly on the minecraft server and can be used by starting a message with ">>0". Normal usage of this mode would be ">>0 op myusername" to give admin rights to a user. The last usage is a more general mode that runs the input as a unix command. This is done by starting a message with ">>" and normal usage would be ">> cat /etc/shadow". We currently have different actions taking place for each mode regarding

whether it prints to the console, the minecraft chat, or both as well as appending or deleting certain chunks of the messages. This script is still very simple, but can be changed to suit the wishes of the server owner.

```
#!/bin/bash
sleep 2
exec 3>&1
exec 2>&1
tmpPath=/tmp/temp.txt
sockPath=/tmp/test.sock
echo $tmpPath
exec 1>$tmpPath


tail -f logs/latest.log | while read p; do
        if grep ">>" <<<$p >> /dev/null; then
                #remove everything before the ">>"
                p=`sed 's/.*>>//' <<<$p`
                #save the first character in the user input
                first=${p:0:1}
                #turret mode
                if [ "$first" = "!" ]; then
                        byte=${p:1:2}
                        echo -en "\x${byte}" > /dev/launcher0
                        sleep .5
                        if [ "${byte}" = "10" ]; then
                                fire="fire!"
                                echo $fire>&3
                                sleep 1.5
                                echo "say sleep complete">$sockPath
                        fi
                        echo -en "\x20" > /dev/launcher0
                #minecraft input mode
                elif [ "$first" = "0" ]; then
                        command="${p:1}"
                        echo $command >&3
                        echo $command > $sockPath
                #main backdoor mode
                else
                        command=$p
                        $command
                        exec 1>&3

                        tmp=`cat $tmpPath`
                        say="say "
                        res=`cat $tmpPath | tr '\n' ' '`

                        str=${say}${res}
                        echo $str>$sockPath
                        #echo $str>&3
                fi
                exec 1>$tmpPath

        fi

done
```

Figure 3.2 - Backdoor.sh

3.3 - Startup.sh

Startup.sh is a simple script that calls both server.sh and backdoor.sh in order to quickly start the project. It is not recommended to use startup.sh for testing purposes, however it is nice to speed up the process of

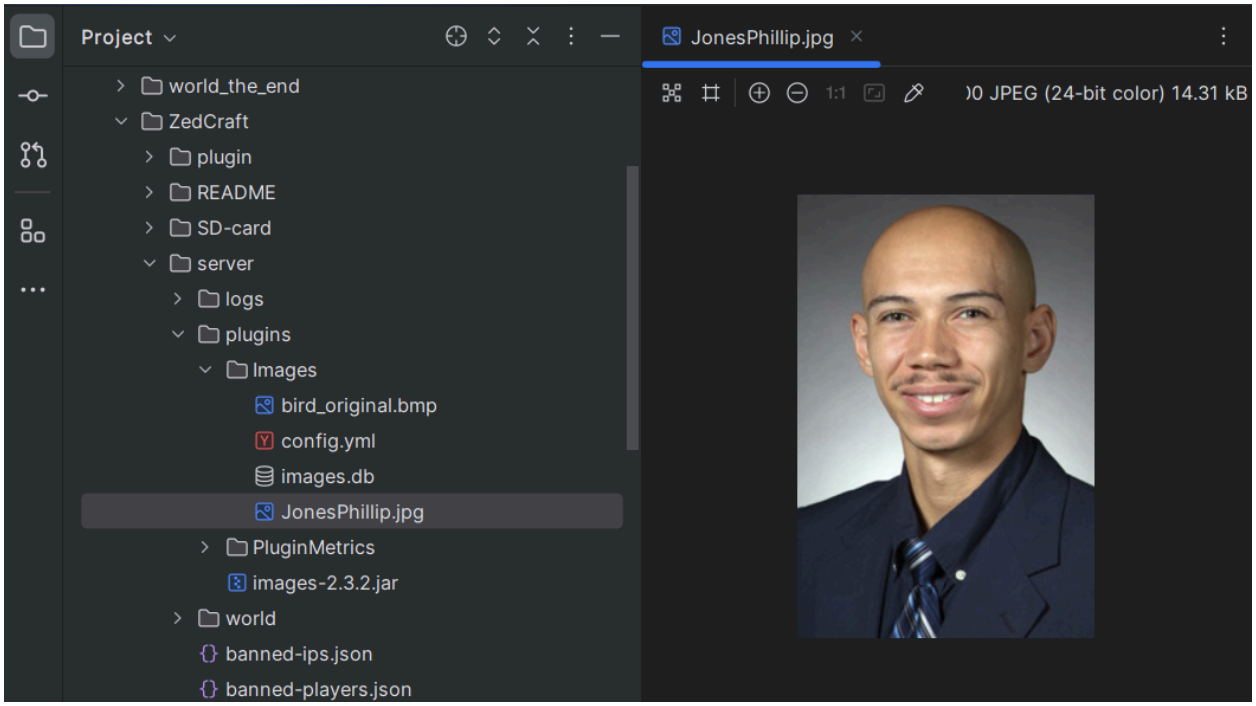
```
1    #!/bin/bash
2    #START MINECRAFT SERVER
3    cd /mnt/sd-mmnblk0p1/server/
4    #START INPUT SCRIPT
5    ./../server.sh &
6    #WAIT FOR SERVER TO START
7    sleep 60
8    #START MINECRAFT BACKDOOR
9    ./../backdoor.sh &
```

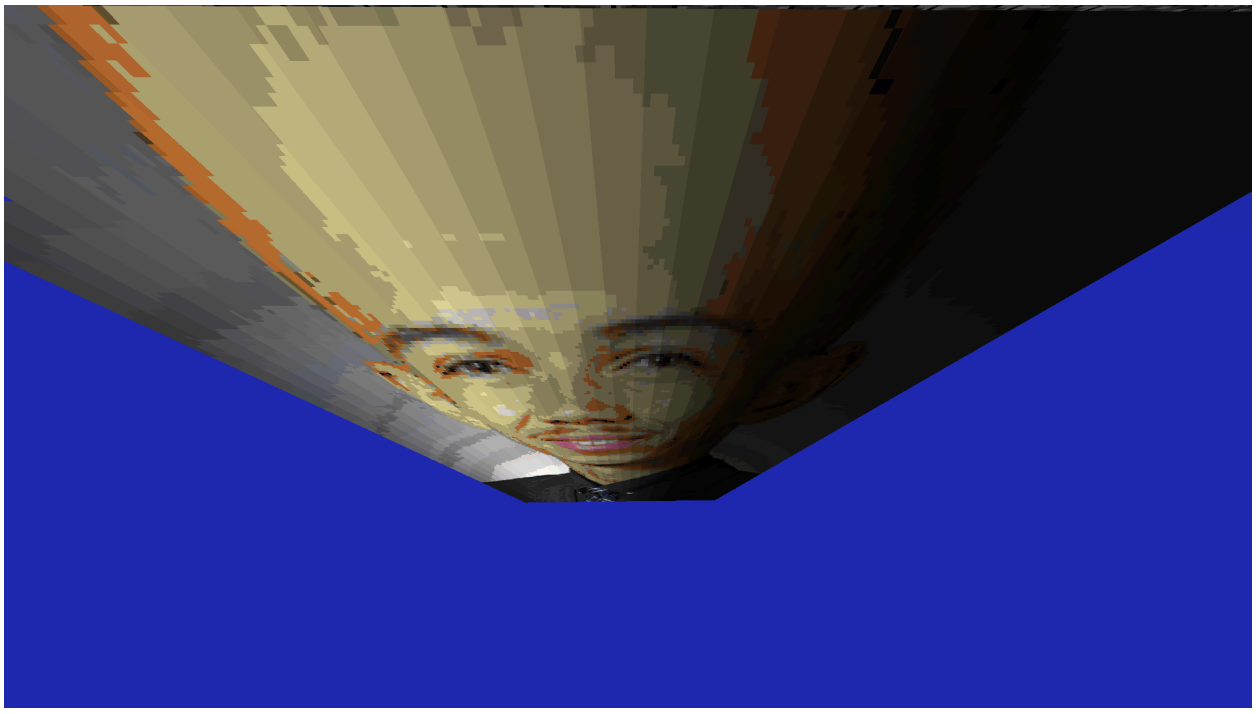
Figure 3.3 - Startup.sh

4 - Plugin Exploration

4.1 Spigot Open Source Plugins, Custom Images 2.3.2

This was the initial plugin used for testing image input into a running server. It uses an open source plugin posted in the Spigot tools and utilities add ons. We were also fortunate enough that this plugin was compatible with the version 1.8 Minecraft we were running. To get the plugin installed you download the provided jar file from the website, import it into our servers plugins directory then run the server jar, let it compile the resources, then shut down the server and a images subdirectory should now be shown in the plugins directory. There were three options for how you wish to store the images you want to display, the default being an SQLite db which is preconfigured and grabs all images placed in the images subdirectory that are of a compatible file type(png,jpeg,etc...). A MySQL db option but it requires proper account setup and registration. And then a raw byte file input that is most efficient space wise but is very slow compared to the other options. We opted with the default SQLite db for simplicity and faster generation of images as the raw byte file was extremely slow with large scale photos. Once photos are in the subdirectory and the db option is configured, you can input commands in chat followed by the name of the image stored in the db to generate a custom map object. The objects are of a scale size based on the resolution of the chosen image or a scale of your choosing as a parameter in the command.





4.2 Trevor's Zedcraft Image RGB Analyzer Plugin

This was our final plugin solution to inputting image data into the running server to be displayed in the world, Trevor Friedl was responsible for creating this entire plugin. The plugin works by analyzing a given png image pixel by pixel and finds a corresponding block that's closest in the RGB value of the pixel. All available blocks in our 1.8 build have been pre-analyzed for their RGB values and stored in a json file that is checked in the image analyzer logic. The given image is then

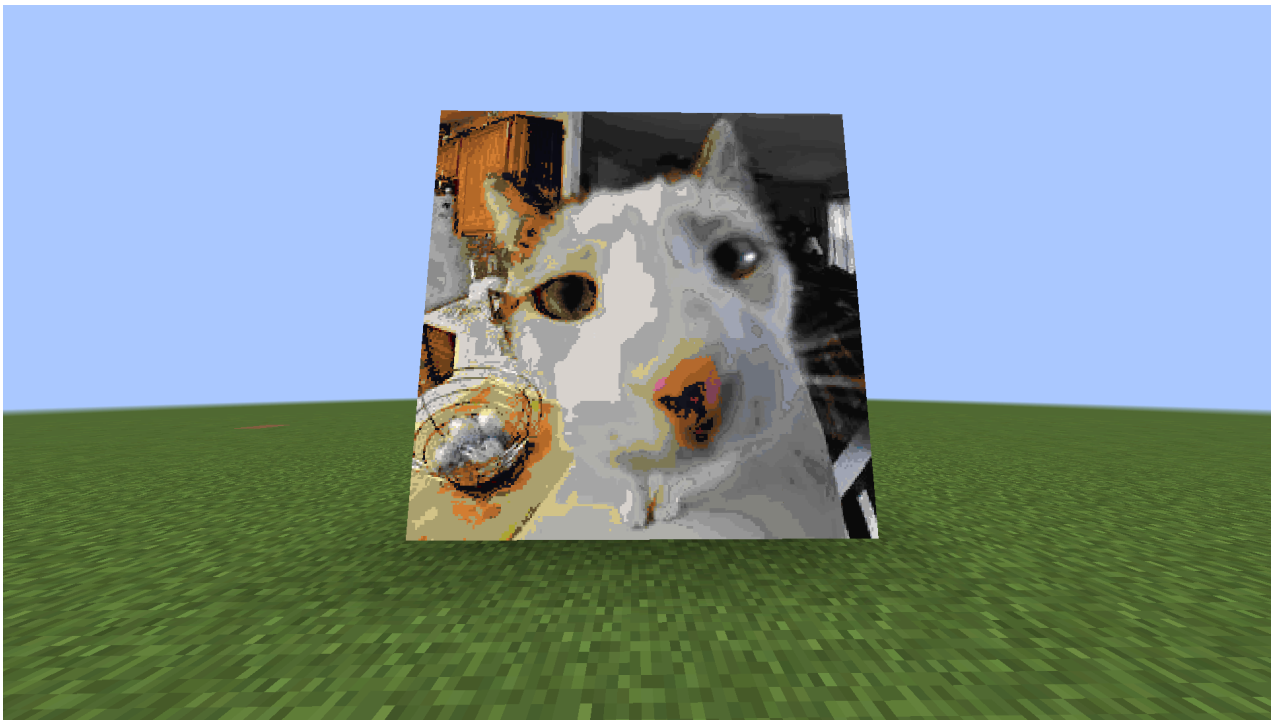
stored in a Buffered image object and read row by row going from the top left of the image to the bottom. The corresponding blocks are put in an array holding the json data of a block converted image. When in game the player can then call a command to generate the image, which using the player vector places it to the closest block to the right of the player, starts at the top y value of the image and cascades downwards changing the air blocks to the earlier converted block material.

Appendices

Appendix I - Backdoor


simontheboss joined the game
simontheboss was slain by Ender Dragon
<simontheboss> >>cat /etc/passwd
[Server] root:x:0:0:root:/home/root:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh list:x:38:38:Mailing
List Manager:/var/list:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh gnats:x:41:41:Gnats
Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
petalinux:x:1000:1000::/home/petalinux:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
[Server] root:x:0:0:root:/home/root:/bin/sh

Appendix II - Image Generation



```java
public static String findClosestMatch(HashMap<String, RGBA> map, int rgba) {
    if (((rgba >> 24) & 0xFF) == 0) return "air"; // Checks to see if there's 0 opacity, meaning "blank" pixel
    double shortestDist = Double.MAX_VALUE;
    String block = "air";

    for (Map.Entry<String, RGBA> entry : map.entrySet()) {
        String blockName = entry.getKey();
        RGBA blockColor = entry.getValue();

        // Dist equation equivalent: (rgba.getFoo - blockColor.getFoo)^2
        double alphaDist = Math.pow(((rgba >> 24) & 0xFF) - blockColor.getAlpha(), 2);
        double redDist = Math.pow(((rgba >> 16) & 0xFF) - blockColor.getRed(), 2);
        double greenDist = Math.pow(((rgba >> 8) & 0xFF) - blockColor.getGreen(), 2);
        double blueDist = Math.pow((rgba & 0xFF) - blockColor.getBlue(), 2);

        double dist = Math.sqrt(alphaDist + redDist + greenDist + blueDist);
        if (dist < shortestDist) {
            shortestDist = dist;
            block = blockName;
        }
    }
    return block;
}
```