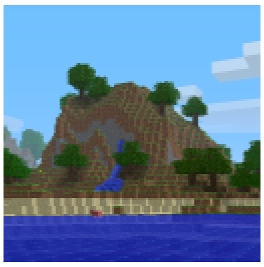
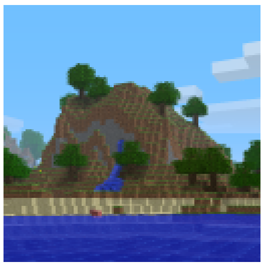


Pack.png

A Search By The Minecraft Community

SalC1 ~~Approved~~ ~~Unofficial~~ Approved


pack.png

This document documents the method used for reversing the seed of pack.png, the default icon of minecraft texture packs and servers.

Brief Summary of what was useful

(~~Full up-to-date methodology coming soon~~)
(Doesn't seem like it lol)

The pack.png seed-finding process was incredibly long and complicated, even discounting the many dead ends along the way. This section is meant to be a quick summary of how the seed was found; for the full methodology, see below (if you're reading this right after the initial announcement, the method below is most likely being updated and may not reflect the actual method used).

To start, we determined the world orientation using the water texture, allowing us to determine the orientation of the player. The cloud texture was then used to find where the positive x axis was, and further confirmed a general approximate world location. After this, a recreation was made to nail down some of the block positions. Various regression algorithms worked alongside the recreation to get block positions accurate, while also refining the exact location and orientation of the player. This, combined with the clouds, eventually resulted in us determining the z-coordinate of the waterfall source block, specifically z=-31.

Once this was established, we used the dirt heights in conjunction with the pattern of dirt/sand on the beach to determine the most likely x coordinates of the waterfall. By far, the most common coordinate that resulted from this was x=116, and since this coordinate kept turning up we decided to go with it. We marked some clearly visible, unambiguous positions of dirt and sand along the beach, and if these were combined with the coordinates of the waterfall it was possible to filter the world seeds. That is, given a certain world seed, we could determine (without opening the game) whether it could potentially be pack.png.

Due to the very large number of potential seeds for randomly generated minecraft worlds (281,474,976,710,656), no one person could check every seed themselves. Instead, we used the compute-sharing platform BOINC to search through this entire seedspace to get a list of around 700,000 possible world seeds. While this may seem like a very large number, it made the rest of the search very easy to complete. Once the candidate seeds were determined, they were run through another program which checked if the height of the terrain matched the recreation of pack.png. This resulted in only one seed by the end: 3257840388504953787

Minecraft@Home Discord invite: https://discord.com/invite/xArErFf

CREDITS: https://mcatho.me/packpngcredits
SalC1's video: https://www.youtube.com/watch?v=ea6py9q46QU
Our Reddit post:
https://www.reddit.com/r/MinecraftAtHome/comments/iocx6f/packpng_seed_was_found_explanation_tutorial_and/

Contents

Version

We have identified the version to be a development version of **alpha 1.2.2a**. It cannot be from alpha 1.2.2a or later, as alpha 1.2.2a was released with the image in the game files. It cannot be from alpha 1.1.2 or earlier, as that update did not contain biomes, and had much brighter leaf colours not seen in the pack.png image. Alpha 1.2.2a was released on November 9th 2010, and according to the file metadata of pack.png, the image was last modified on November 8th 2010.

Reading the code and modding

Reading the code

The recommended way to read the code is to set up the alpha example mod. To do this you will need:

- A copy of JDK8 or newer.
- A copy of IntelliJ IDEA. The Community edition is free for everyone and very good, but the Ultimate edition is better and free with a student license.

To set up the example mod, follow these steps:

1. Clone/download the alpha example mod repository.
2. Open the folder you cloned/downloaded and open the command prompt/terminal. In Windows run the command “gradlew genSources”, or in Linux/MacOS run “./gradlew genSources”. Wait for the command to finish.
3. Open IntelliJ IDEA, click “Import Project”, select the folder you cloned/downloaded, go to the next screen and select “gradle” from the project types. Leave all the rest of the settings on default and click finish. Wait for the project to finish importing.
4. The Minecraft alpha sources can be found in “External libraries” -> “net.minecraft:minecraft:blah blah” -> “minecraft blah.jar” -> “net.minecraft”.
5. Open any Minecraft class, and click “attach sources” on the blue banner at the top of the editor. It should automatically open to the location of the jar file; select the sources jar directly above it and press OK.

Useful shortcuts in IntelliJ IDEA:

- Ctrl+N: jump to Java file by name (press ctrl+N again to toggle between only showing files from your mod and showing all files).
- Ctrl+left click on a reference: jump to declaration of method/field/variable/class.
- Ctrl+left click on a declaration: find references of method/field/variable/class.

Sections in this document may contain references to parts of the code, marked in footnotes.

Modding

First, follow the steps for reading the code. Then, it's the same as writing a normal Fabric mod, except remember there is no Fabric API, only the Fabric loader. For help with Fabric modding in general, ask in the Fabric discord (https://discord.gg/v6v4pMv), **but remember they have a rule against mentioning MCP names there, so don't share any alpha class/field/method names there**. For mixin help, you can ask in the #mixin channel in the Sponge discord (https://discord.gg/sponge).

Basics of world generation in alpha

The world seed

There was no way to choose a world seed in alpha 1.2.2a, all world seeds were random. World seeds have a data type of "long", which means they can be in the range -2^63 to 2^63-1 inclusive, so at first glance it looks like there should be 2^64 possible seeds. However, random seeds are generated using the following code:

`worldSeed = new Random().nextLong();`[1]

Java's Random object uses an RNG with only a 48-bit seed internally, meaning that there are only 2^48 possible outputs of nextLong. Moreover, some of these outputs are repeated, meaning that in actual fact there are only 2^48 * 0.82 possible seeds, or about 231 trillion.

System time (spoiler: this got us nowhere)

How "`new Random()`" initializes its seed is platform and Java version dependent. Notch confirmed that they were using Java 6 and Windows 7. In Java 6, the seed was initialized as follows:

`setSeed(++seedUniquifier + System.nanoTime());`[2]

Where "seedUniquifier" is a global variable that starts at 0 on program startup and increments each time a Random object is created.

`System.nanoTime()` returns a nanosecond-precision counter relative to an arbitrary point in time. On Windows 7, this returned the number of nanoseconds since system startup, except in 2010 most system clocks weren't as advanced as today, and only had microsecond accuracy, which meant that until it overflowed (after a long time), `System.nanoTime()` would always be a multiple of 1000, and once it did overflow, it would still be a multiple of 8 (the greatest common divisor between 1000 and 2^64). However, since Random seeds are only 48-bit, the lowest 48 bits of `System.nanoTime()` would overflow sooner, after 2^48 nanoseconds or approximately 3 days. According to a Tumblr post by Notch from around the time, he tended to leave his computer on so it may well have overflowed. If the pack.png world is the first world to be created after game

[1] `World.World(File, String)`
[2] `Random.Random() [JDK 6]`

startup, then the Random which generated the world seed is the 17th Random to have been created, which would make the lowest 3 bits of the seed 001. However, this is a risky assumption to make, as multiple worlds may have been created. If it wasn't the first world, then we can make no accurate guess on any bits of the seed using the system time.

World generation

Chunks are generated in 2 stages: "generation" and then "population".

Generation[3] handles the basic terrain: its height, shape, the top decoration and caves. The shape of the terrain[4] is decided by perlin noise generators seeded based on the world seed. Decoration[5] refers to the top blocks of the terrain, usually grass and dirt and water below sea level, but sometimes sand and gravel and other blocks too, and the bedrock underground. The RNG used for this step is *independent from the world seed* and only dependent on location, which is why bedrock is the same in every world; however, some things in this step also use perlin noise generators which are dependent on the world seed, so we can't tell location from the sand on the beaches, for example. This independent from world seed fact will be used later on. Caves are seeded in the same way as population[6], but cannot really be used as it is hard to tell from just the pack.png image where any caves start generating.

Population[7] handles the more varied aspects of world generation, such as trees, flowers, dungeons and ores (and in later versions, structures). Population occurs the first time a 2x2 area of chunks is loaded, in the 16x16 block region in the middle of them (therefore offset by 8 blocks). This is done so that trees and other features don't get cut off on chunk borders. This 16x16 region offset by 8 blocks is referred to as the "population region".

Almost the entirety of population is done using a single Random object seeded from the world seed and location of the region as follows:

```
rand.setSeed(worldSeed);
long a = rand.nextLong() / 2 * 2 + 1;
long b = rand.nextLong() / 2 * 2 + 1;
rand.setSeed((chunkX * a + chunkZ * b) ^ worldSeed);
```

Where ^ is the bitwise xor operator, and / is the integer division operator, which always removes the decimal part (i.e. rounding towards zero). The seed of the Random after this point is referred to as the "population seed" or "chunk seed".
Chunk seeds are important because there are only 2^48 of them, they are a way to gain information about the seed without explicitly knowing the location. Instead of cracking the world seed directly, we instead crack the seed at this point, and work backwards from there.

[3] `ChunkProviderGenerate.func_533_b(int, int)`
[4] `ChunkProviderGenerate.func_4060_a(int, int, byte[], MobSpawnerBase[], double[])`
[5] `ChunkProviderGenerate.func_4062_a(int, int, byte[], MobSpawnerBase[])`
[6] `MapGenBase.func_867_a(IChunkProvider, World, int, int, byte[])`, inherited by `MapGenCaves`
[7] `ChunkProviderGenerate.populate(IChunkProvider, int, int)`

A methodology for converting a chunk seed to a world seed given a guess of location will be shown later in this document.

Tree generation

There are a couple of annoying tree-related things in population which use simplex/perlin noise instead. One is biomes, which affects the number and types of trees which generate, and the other is what we call “tree density noise” or just “tree noise”[8], which is a perlin noise generator that also affects the number of trees which generate in a population region.
In rainforests, large trees can randomly generate where they otherwise couldn’t, which is both an extra random call (throwing off seed-reversal-gpu and seed-tester-native), and breaks one of our assumptions for the Z coordinate too (although we would still be very confident on that value). By observation of the image and foliage colour, we have concluded that the mountain is not in a rainforest biome.
Other than that, the number of tree attempts in a chunk is based on the tree density noise, with a biome-dependent constant added to it. The theoretical maximum number of tree attempts is actually very high, and if we went all the way to this maximum our search would be too slow. Therefore, we picked a lower maximum number of tree attempts of 12 which it is almost certainly below. To decide on this number, we simulated a large number of chunks for each biome:

Biome	**Number of chunks tested**	**Total tree attempts in sample**	**Min. tree attempts in sample**	**Mean tree attempts in sample**	**Max. tree attempts in sample**	**Standard deviation**
Rainforest	611	4356	2	7.129296	13	1.712436
Swampland	905	96	0	0.106077	1	0.308107
Seasonal Forest	2856	11191	-1	3.918417	9	1.575646
Forest	10130	68557	2	6.767720	13	1.540529
Savanna	2756	276	0	0.100145	1	0.300248
Shrubland	3037	299	0	0.098452	1	0.297974
Taiga	2019	13621	3	6.746409	12	1.550626
Desert	2835	-56421	-20	-19.90159	-19	0.297924
Plains	2024	-40267	-20	-19.89476	-19	0.306935
Tundra	3021	-60139	-20	-19.90698	-19	0.290502

The number of tree attempts can be negative, which is the same as 0 tree attempts.

[8] `ChunkProviderGenerate.field_920_c`

Recreation

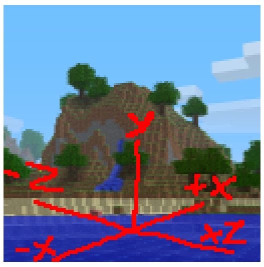
As of writing, there have been at least two major recreation-building efforts (in addition to several spinoff attempts). Between these two attempts, major improvements were made to the techniques used:

- Use of software and bots to provide visual overlays to check the accuracy of the recreation.
- Use of programming and 3D projection math to precisely estimate the player's relative position/orientation.
- Additional discoveries about the image.

As a result, a lot of older estimates have been revised, leading to significantly better accuracy.

Orientation

One of the first things to be found was the orientation of the player. This was found using the texture of the water. This was found by Gsmack and Braycam2; when he realized that streaks in the water's texture could help us identify the player's location. The Z axis is left-right (increasing right), and the X axis is front-back (increasing forwards). It is actually not possible to tell whether the player is facing towards +X or -X, they just got lucky; we later confirmed the direction with the clouds.



Location

Most of the time we will define the location of things in terms of the coordinates of the water source block in the waterfall, as that's an obvious fixed point on the mountain that stands out. This section will focus on trying to find the location of the waterfall in the world, and everything else can lead on from there.

Y coordinate

This is the easiest of the 3 axes to find, as sea level is constant at y = 64[9] (note this was decreased to 63 in a later version). Counting up from the sea level, we get a value of **waterfall Y = 76**.

[9] `ChunkProviderGenerate.func_4062_a(int, int, byte[], MobSpawnerBase[])`

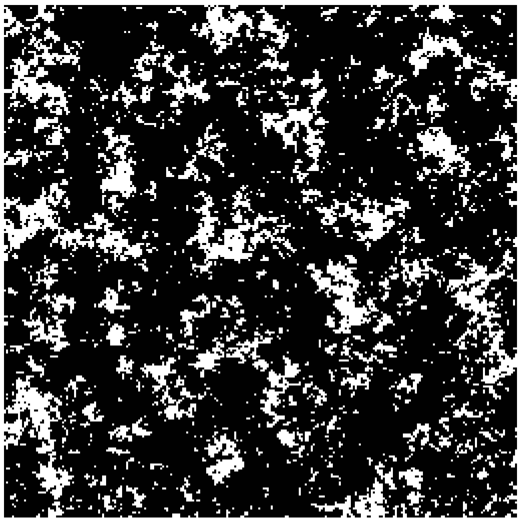
(Additional note: Here, “sea level” of 64 refers to the boundary between water and air. Since the y-level of a block is measured by its bottom surface, this is the same as saying the uppermost water blocks are at y = 63 and the lowermost air blocks directly above sea level are at y = 64).

Clouds

The first breakthrough for finding the location of the mountain was identifying that the cloud pattern is the same every time you start up the game. From the code[10] and experimentation, we identified the following key information about clouds in alpha 1.2.2a:

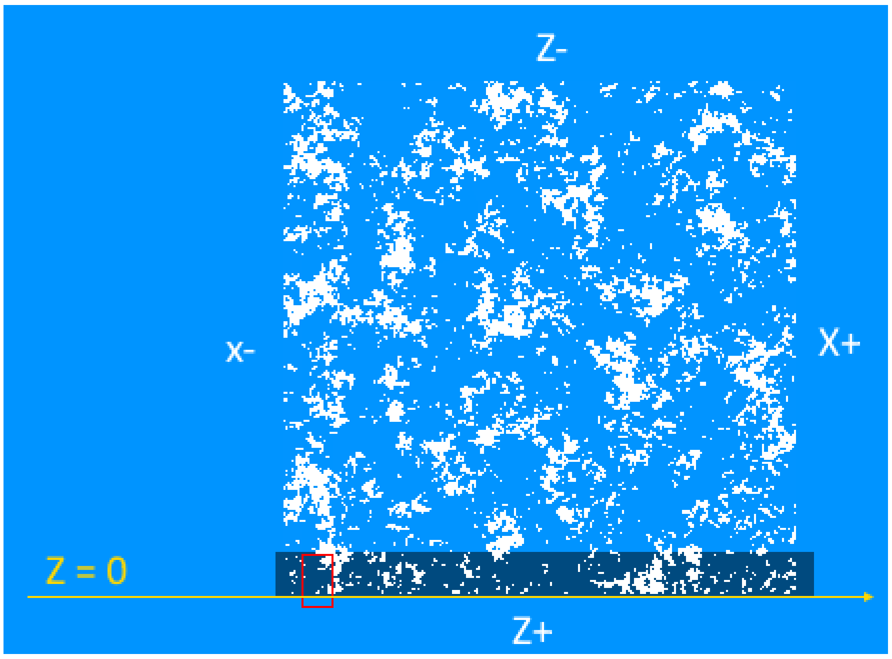
- The cloud tile layout is taken from an image in the game files called “clouds.png” (depicted below).
- Clouds move towards -X, but not in the Z axis.
- Clouds move at a rate of 0.03 blocks per tick, or 0.6 blocks per second, but only when a world is open. They do not reset their position when closing and opening a world, but they do reset after restarting the game.
- Clouds are a different size depending on whether you are using fancy or fast graphics. Clouds do not have a depth in fast graphics[11], so pack.png was taken with fancy graphics.
- With fancy graphics, a cloud tile is 12x12 blocks in size. Since clouds.png is 256x256px, clouds repeat every 3072 blocks.
- The bottom of the clouds are at y = 108.33, and have a height of 4 blocks.
- At time 0, the corners of the clouds.png image is aligned to block coordinates (0, -4). Thus being offset towards -Z by 4 blocks for some unknown reason.

Clouds.png (with transparent pixels changed to black so you can see it):

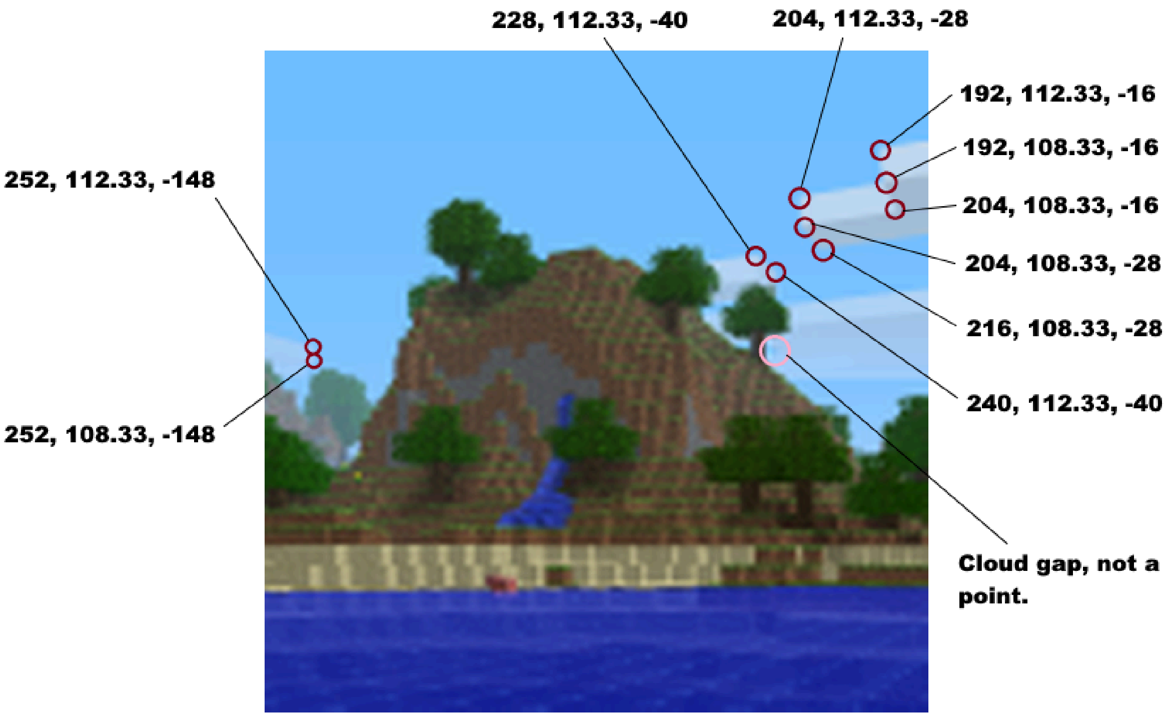


[10] `RenderGlobal.func_6510_c(float)`
[11] `RenderGlobal.func_4141_b(float)`

We identified the location of pack.png to be here on clouds.png, near the bottom left.



The coordinates of all of the clouds at time 0 could then be determined:



However, the time elapsed in-game would need to be found in order to get the x-coordinates of these points, as again, clouds move towards the -x direction.

Perspective fitting

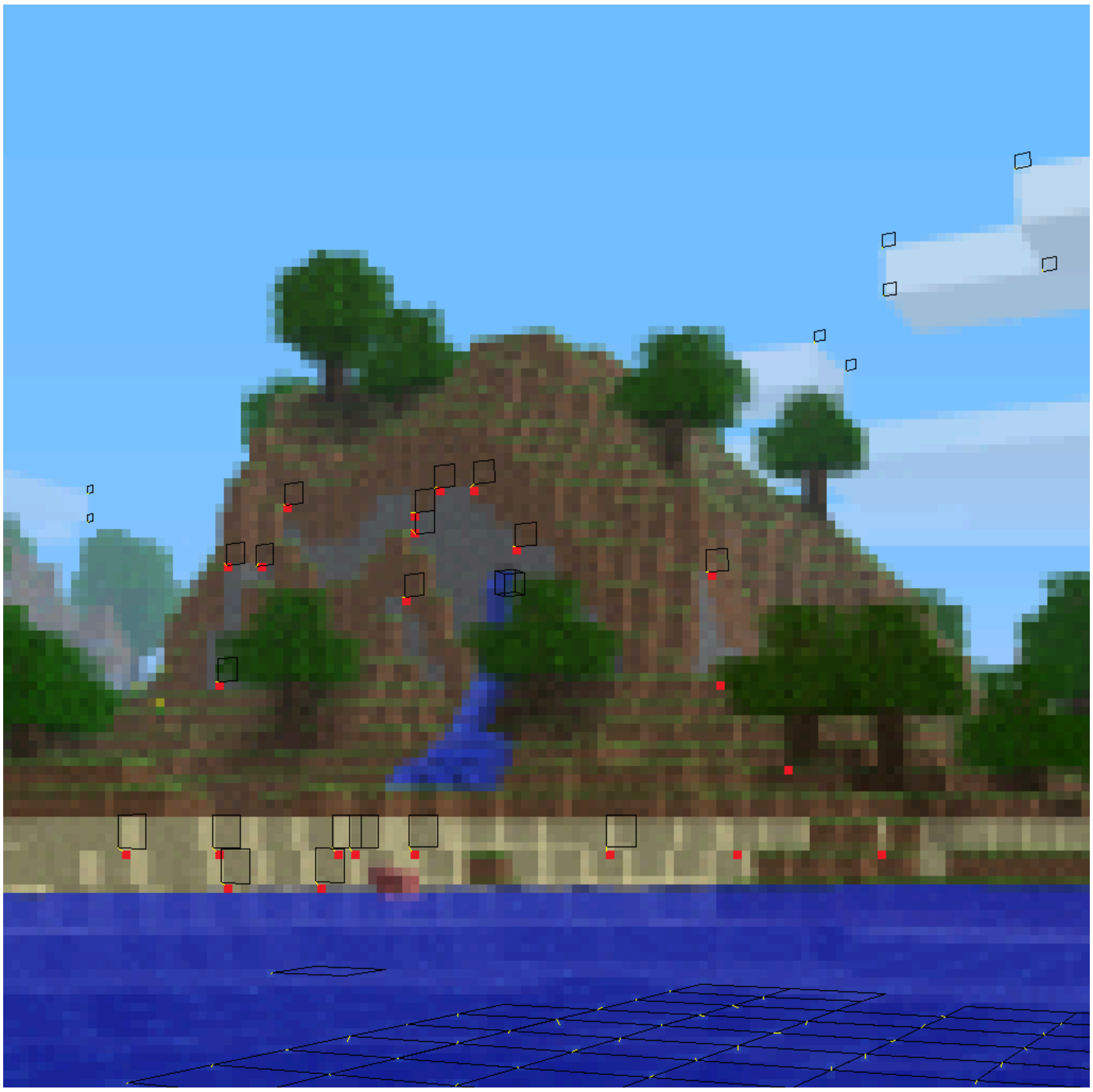
Finding the player’s position and orientation can be viewed as a sort of regression problem. By taking advantage of known 3D relationships that are visible in the image, it is possible to get relatively precise estimates.

Four main components were used to determine perspective:

- The clearly visible front face of the hill.
- Clearly visible sand blocks on the beach.
- Clouds.
- The water texture.

Various algorithms were used, and many different parameters were varied in this optimization process. These included not only the player’s position/orientation, but also any unknown X/Z offsets between the various components. Additionally, the screen resolution and the section of the screen which was cropped were needed. This helped to determine the player’s position relative to the clouds, and the hill position relative to the player. When combined, these gave the position of the waterfall relative to the clouds, which allowed the z-coordinate of the waterfall to be determined.

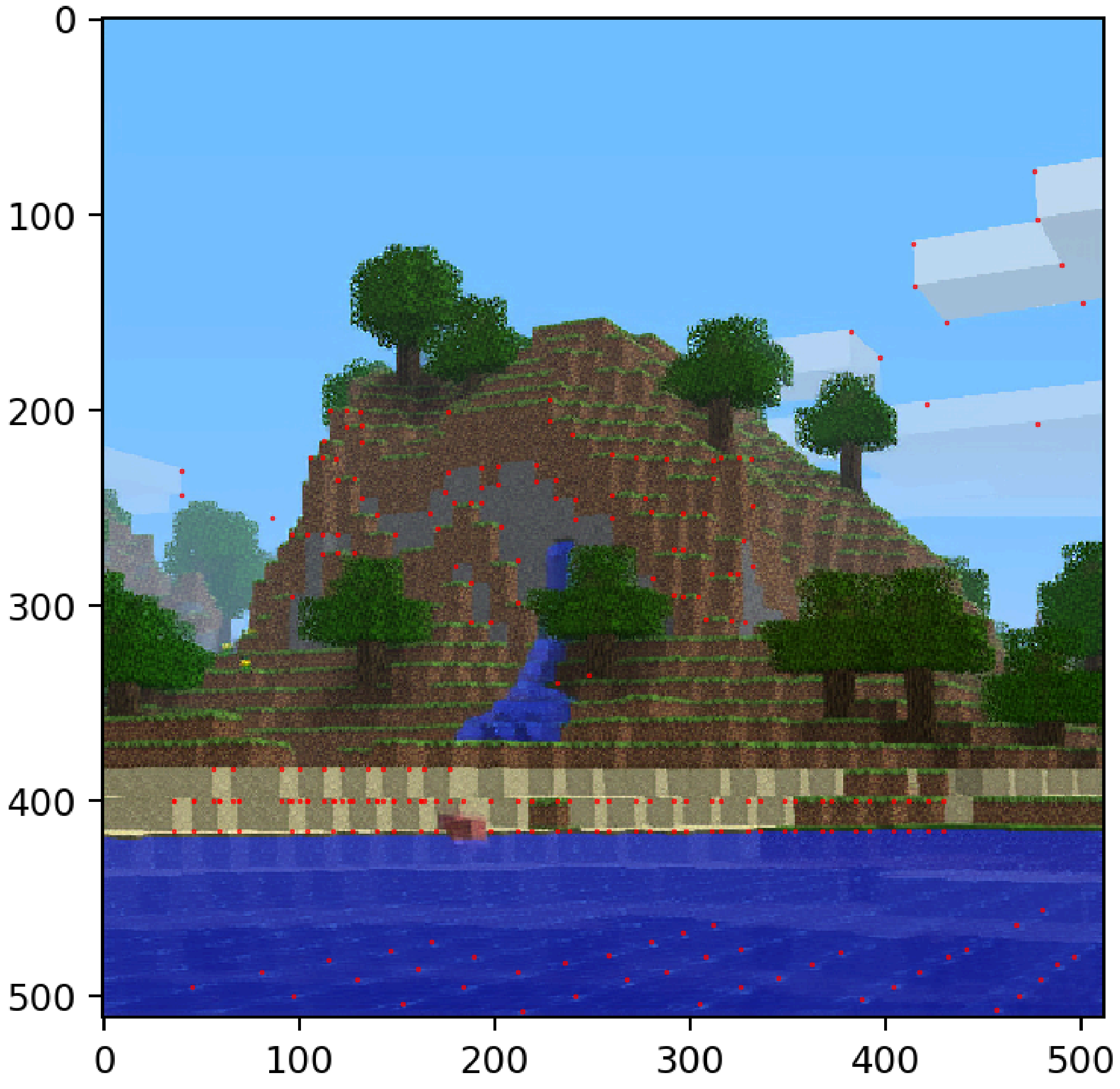
Example render from an (older) perspective fit:



These following estimates were originally obtained:

- Player feet position relative to the waterfall source block is:
  - Most likely -68 to -69 on the X axis
  - Definitely -12 on the Y axis (player is standing on a shore at sea level)
  - Most likely between +32 and +33 on the Z axis.
- Player orientation is:
  - Facing east with a yaw between -119 and -121
  - Looking slightly upwards with a pitch between -1 and -3

This fit was later refined as the recreation got better, and more points were used on the original image. The points used can be seen below:



With the use of more points, the fit became much more accurate, and made some errors in the recreation apparent - most importantly, the position of the beach had to be moved slightly backwards.

After visually altering the parameters slightly, a final, very accurate set of coordinates was found:

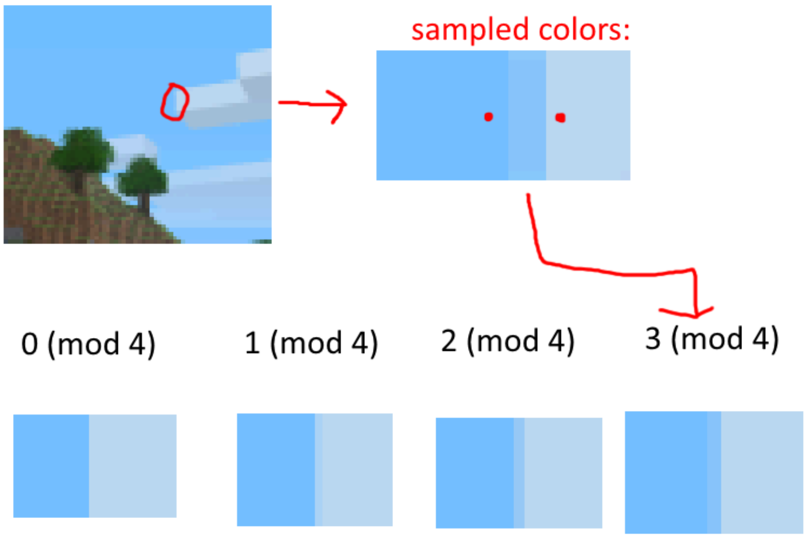
- Player is offset relative to the waterfall by (x = -66.84, y = -10.38, z = 31.72)
- The pitch is -8.297 and the yaw is -119.23
- **The global z coordinate is z = -31 (based on the cloud position)**
- The screen resolution is 1600x1116, with the original crop starting at pixel (622,284) (the full crop being 512x512, as explained below)

This was a very important development, as it gave the z-coordinate for the waterfall.

Determining original resolution and cropping

The image metadata clearly states that it was edited in Paint.NET v3.5.5. This version of the software only has a few options for downscaling algorithms. It was determined that the default algorithm (labeled “best quality” or “supersampling”) was a “box”-style algorithm, and upon visual comparison it was determined that this was the downscaling algorithm used.

A major breakthrough occurred while studying the pixelated borders of the clouds. It was discovered that the color blending perfectly matched that produced from a 4x downscale.

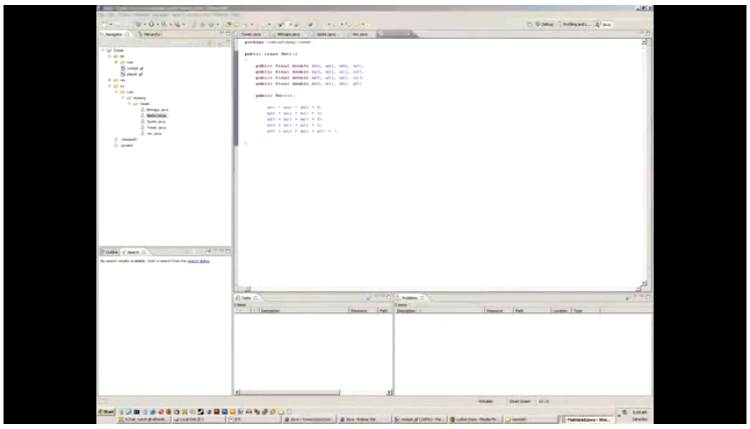


custom downscales of just the 2 sampled colors next to each other

Furthermore, the presence of many “sharp” edges with no blending also hints that the downscale ratio was a perfect integer. As a result, we can conclude that the original cropped image, before downscaling, was 512x512 pixels. Each pixel in the final image is simply the average value of the 4x4=16 corresponding pixels in the original.

The vertical FOV of minecraft was hardcoded to be 70, but the vertical/horizontal FOVs of the cropped region must be estimated. Most importantly, the cropping/FOV estimates produced during the perspective matching process allow us to determine where the player’s crosshairs would be if they were visible.

It also becomes possible to estimate the original vertical resolution of the uncropped screenshot. The resulting value is in the neighborhood of 1125-1145 pixels. At first this value may seem unreasonably large, but it is in line with what would be produced by a maximized window on a 1600x1200 monitor, once the taskbar and title bar heights are subtracted (one such test produced 1138 pixels). Below is an early screenshot (a couple years before pack.png) from Notch which indicates he likely owned a 1600x1200 monitor.



Unfortunately, no screenshot of him playing minecraft on such a monitor has been found. Regardless, this mainly serves to confirm that the estimates of resolution and FOV are viable. After exploring various settings/configurations, **1600x1138 is considered the most likely original resolution**. It is unlikely that a better estimate of vertical resolution could be found.

Upscaling attempts

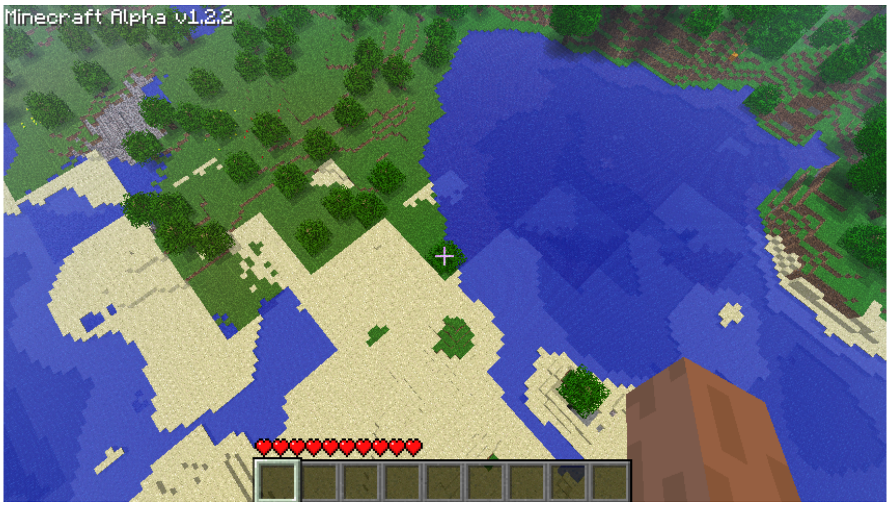
There have been a few attempts to upscale the image. The first attempts to upscale the image, using “AI”, produced results that looked nice but distorted the positions of the block edges too much to really be usable.

Since the downscaling algorithm is very simple, it has been possible to manually “upscale” some of the simple parts of the image (such as the borders of the clouds) to the original 512x512 resolution, with the help of code to determine possible color combinations. This in turn helps to greatly improve the accuracy of the player position/orientation estimates.

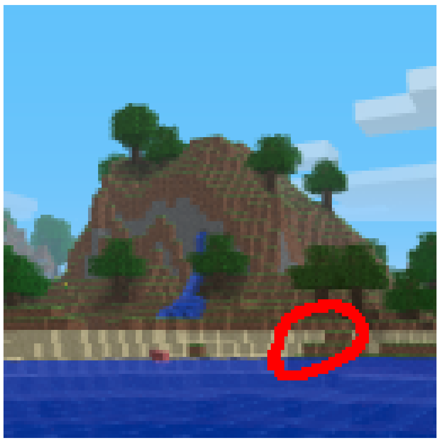
Now that much more is known about the original resolution, orientation, and cropping, there is currently a second wave of attempts at producing a more accurate AI upscale. It should be noted, however, that seed-finding generally favors the use of a smaller number of unambiguous landmarks (like the tree positions easily seen in the raw image). Thus a higher quality AI upscale, perhaps rather unintuitively, is unlikely to be of significant benefit in this regard.

Chunk borders

To get exact coordinates, it would be helpful if we could find the chunk borders. The first major observation involves chunk borders showing between sand and grass on beaches. It is unknown exactly why this occurs.



It was initially believed that the effect was demonstrated in the pack.png image as well, which would tell us the Z chunk border.



Tracing this back to the waterfall and aligning our previous estimate from the clouds with it, this gets us a value of waterfall Z = -30. According to the most current recreation, however, this border appears to have been an optical illusion: the two grass blocks are not actually aligned.

The best estimate for waterfall position (as per the fit above) is Z=-31 based on alignment of the clouds, and this ended up being correct. As the estimate of player position changes, this estimate may also change, but the current value is believed to be +/- 1 block at the very worst.

Another thing that backs up the Z coordinate at least approximately is the tree types. In alpha 1.2.2a, it is not possible to have a small tree and a big tree in the same population region (except in rainforests). On the top left of the hill, you can see a big tree very close to a small tree, which means there must be a population border between them. There is also a similar case on the right of the image. This is not hard proof of the chunk border, but can serve to narrow down the possibilities. Note: the precise locations of these trees were not completely determined.

X/Z coordinates from the clouds

Using a mesh from an early approximate reconstruction of pack.png imported into Blender, and the pack.png image overlayed, an approximate 3d reconstruction of the image was created, and the cloud edge was traced down to give an approximate Z coordinate of the waterfall of about Z = -31 to Z = -29 <I just made up these numbers but it was approximate like this. TODO look up our actual estimate from the time>. The most up-to-date reconstruction and perspective gives Z = -31.

The approximate X coordinate of the waterfall as a function of time *t* seconds of ingame time since game startup is *x(t) ≈ 134 - 0.6t*. We are starting by assuming that Notch took between 30 seconds and 5 minutes to take the screenshot, which gets a range of X coordinates of 116 down to -46. (Note: an earlier estimate was *x(t) ≈ 196.61 - 0.6t* to give a range of 178.61 down to 16.61, simplified to 0-180.)

Biomes

It is possible to extract a small amount of climate-related data. For example, the hue of the sky places a lower bound on the temperature at the block where the player is standing. Furthermore, a slight color gradient visible in the tree foliage establishes the direction of the climate noise at that area. It is not yet decided to what extent this information will be used in the search.

Cloud clipping and sky gradient angles

There is a visible horizontal line where the clouds are clipped at maximum render distance (set to FAR, giving a clipping plane 256 blocks away, oriented with the player head). There are also horizontal lines visible as slight color transitions in the sky. The angles to these lines can be determined (in the case of the cloud clip, it's 9.5 +/- 0.1 degrees) and this can be used to estimate player distance from the hill. Unfortunately this method by itself only gives a +/- 3 block distance estimate which is not as accurate as other methods.

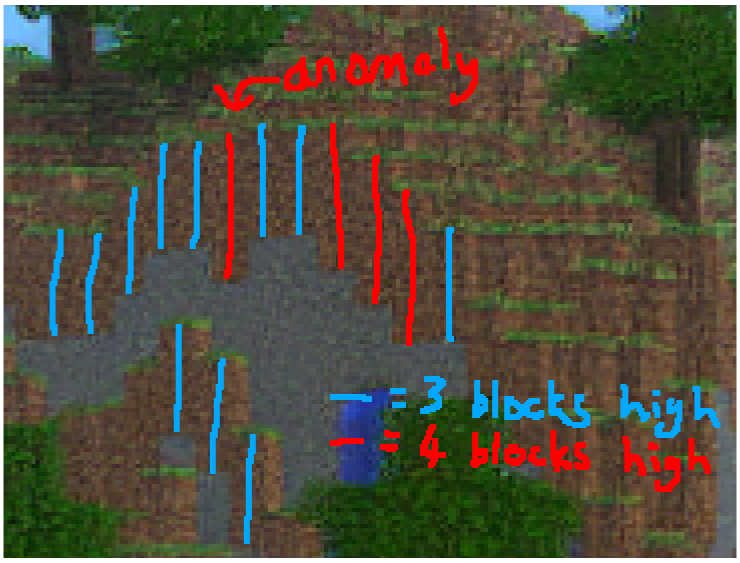
Dirt anomaly

Remember when we said earlier that the generation RNG is defined in terms of the location but not the world seed? Well, that fact can be used to narrow down possible X-coordinates for the waterfall. In alpha 1.2.2a, the depth of the dirt underneath the surface is variable, and is clearly visible in the hill in the pack.png image.
The height of the dirt at a given x and z coordinate is determined by:

```
dirtHeight = floor((dirtNoise(x, z) / 3 + 3) + (rand.nextDouble()
* 0.25));
```
[12]

[12] `ChunkProviderGenerate.func_4062_a(int, int, byte[], MobSpawnerBase[]), ChunkProviderGenerate.field_908_o, ChunkProviderGenerate.field_903_t`

Notice there are two sections to this formula. The first section which depends on a perlin noise generator for dirt height, which depends on the world seed and location; and the second section which does not depend on the world seed, only the location. The key observation is the perlin noise generator produces a smoothed out curve of dirt depth, and does not produce abrupt changes which would require an abrupt change in the rate of change of dirt depth. Any abrupt change like this you see in the world must be caused by the RNG independent from the world seed.
The image below is a sample of dirt heights from the front of the mountain in pack.png. You can see the general trend in the Z direction here is that dirt depth is increasing from 3 to 4 from left to right. However one column of dirt seems to ignore this trend, which means it must have got some extra height from the RNG compared to the block to its right. Additionally, X coordinates where the value from the RNG at the anomaly is higher than the value to the left are more likely, although we didn't end up using this fact as it's a bit too risky.



This allowed us to immediately eliminate half of the possible world X coordinates of the waterfall.

The code used for filtering out x coordinates can be found here:
https://github.com/pack-png-mods/seed-tester/blob/master/src/me/balint/XFilter.java

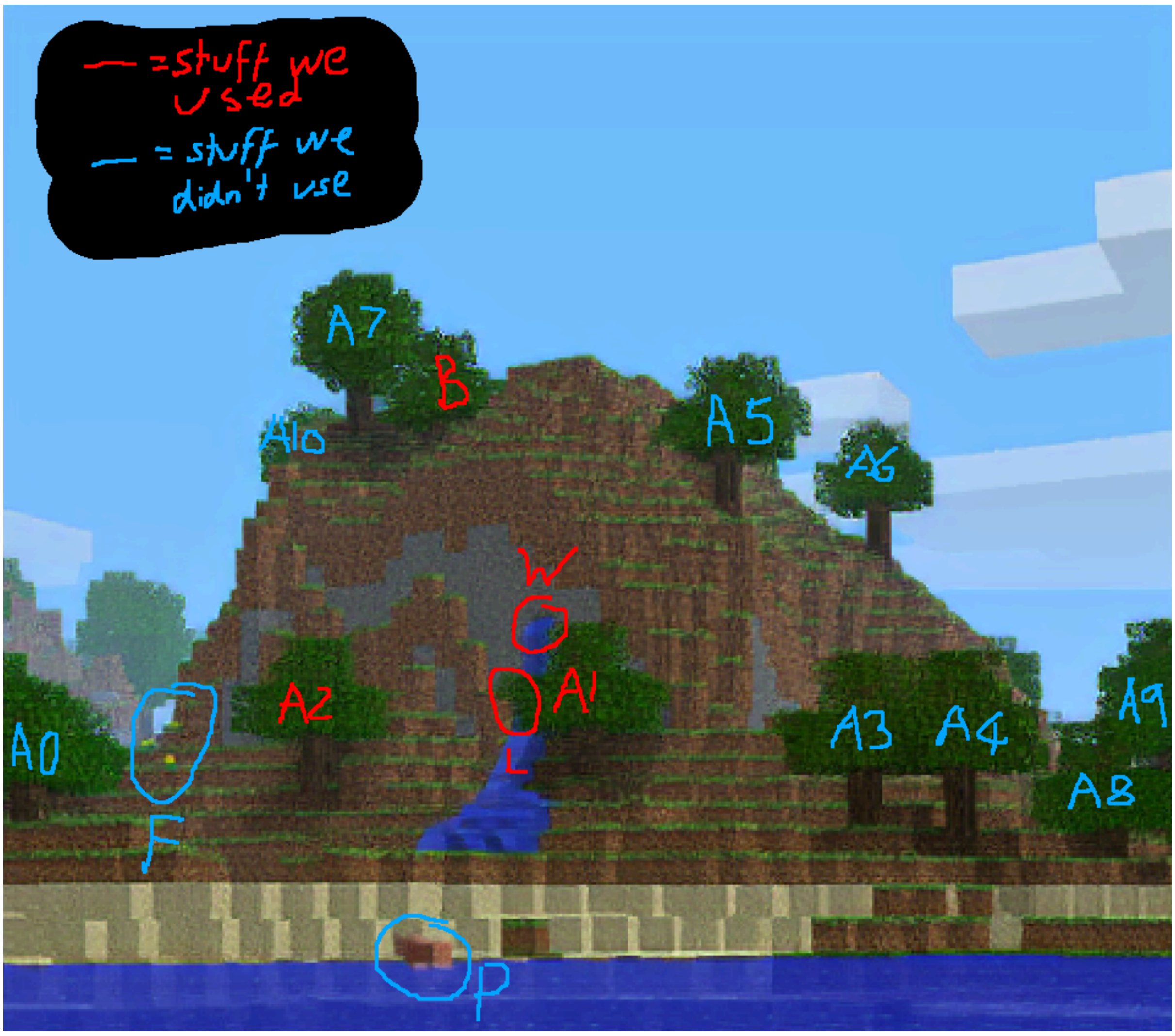
Reversing chunk seeds (legacy section)

Important notes about the following sections of the document:

- It deals with a work-backwards approach of finding possible chunk seeds, then deriving world seeds from those. The newer approach will likely be working forwards, performing a more direct search over world seeds.
- It uses some older estimates for X and Z coordinates.
- The text currently remains how it was originally written.

Refer back to the "world generation" section to remind yourself what a chunk seed is, if you're forgotten.

What can we use to reverse the chunk seed?



Let's tackle these things one by one:

- P: although in modern versions, animals can spawn with terrain, in alpha 1.2.2a they couldn't, therefore this pig does not give us any information on the seed.
- F: flowers are added during population, but they spawn in patches[13], and it is hard to know the exact point where the patch started generating from, which we would need to know for reversal.
- W: a waterfall is a very good candidate for reversing a chunk seed, especially one this high up where they are quite rare; a waterfall has a 0.4257% chance of spawning at y = 76. We used this waterfall where we could, and used things in the same population region as it. There are 50 attempts at a waterfall per chunk, which means we would expect 21.3% of chunks to have at least one waterfall attempt at y = 76. However most of the time a waterfall attempt fails, as it needs to be inside stone, have 5 stone around it, and exactly 1 air block adjacent to it[14].

[13] `WorldGenFlowers.generate(World, Random, int, int, int)`
[14] `WorldGenLiquids.generate(World, Random, int, int, int)`

- A0, A3, A4, A5, A6, A7, A8, A9, A10: all these trees cannot be in the same population region as the waterfall, because they are across a population border in the Z axis. A5 is particularly on the edge, only just over the population border.
- A1, A2: these trees are close to the waterfall so were likely to be in range of it, and were used as part of the reversal process. The only thing about the location that matters about these trees is their x and z coordinates within the population region. A1 is at (-3, 3) relative to the waterfall, and A2 is at (-5, -8) relative to the waterfall. There is also a mathematical technique explained later to narrow down the seeds using a tree location without brute forcing.
- B: this tree is hard to make out exactly where it is on the image, so it earned the nickname “blobby tree”, because it looks like not much more than a blob. However, even it’s approximate location is useful information that can be used to narrow down the chunk seeds. We determined it to be (3 to 5, -9 to -6) relative to the waterfall.
- L: most of the leaves in the image are too ambiguous for us to rely upon, however these two are pretty obvious, so we used them.

Splitting up the X coordinates: “x0,x1,x2..x14,x15”

To reverse chunk seeds, we need to know the coordinates of the features inside the chunk relative to the population region. However, we don’t know this information for the x coordinate (the narrowing down of the global X coordinate using the dirt anomaly does not help with this). So, we unfortunately had to try all 16 possible X alignments.
We named these X alignments after where the waterfall is relative to the start of the population region. Thus, x2 refers to the case where the waterfall X is 10 (mod 16), because 10 = 2 + 8, because population regions are offset by 8 from chunk borders.
In some x alignments, there are some trees in range of the waterfall, and in others, different ones are:

X alignment	Case
x0 to x2	A1 and A2 are not in range of the waterfall, blobby is in range. We search without the waterfall, but with A1 and A2.
x3 to x4	A1 and blobby are in range of the waterfall, but A2 is not. We search with A1, blobby and the waterfall.
x5 to x10	A1, A2 and blobby are in range of the waterfall, we search with all of them.
x11 to x12	A1 and A2 are in range, but it is ambiguous whether blobby is in range of the waterfall. We split these into the case where blobby is in range and blobby isn’t in range. These are named x11_blobby and x12_blobby for the former case and simply x11 and x12 for the latter case.
x13 to x15	A1 and A2 are in range, but blobby is not in range, we use A1, A2 and the waterfall.

Lattices: mathematical reversal with trees

The lattice technique exploits a weakness in the type of random number generator Java uses to generate its pseudo-random values. In simple terms, it’s a fast (and parallelizable) method to enumerate all possible seeds where 2 consecutive random calls produce a given output. Explaining it in full here would take up several pages on its own, so let me instead link an excellent source describing the method in detail:
https://nbviewer.jupyter.org/gist/EDDxample/38a9acddcd29f15af034fd91da93b8fa

Seed-reversal-gpu

https://github.com/pack-png-mods/seed-reversal-gpu
Seed-reversal-gpu is the first step in the pipeline for generating a list of possible chunk seeds. The GitHub repository has a branch for each x alignment. The way it works is as follows:

- Feed the position of A2 (x0-x2 and x5-x15) or A1 (x3-x4) into a lattice to obtain a list of initial seed guesses. There is some weird manipulation to convert the GPU thread ID and offset into a point on the parallelogram, but it is otherwise the same concept.
- Filter out seeds where the height of that tree does not match what is seen on pack.png.
- Since this tree may not have generated at the start of tree generation, iterate over the possible numbers of calls between the start of tree generation and the generation of this tree. A tree attempt does 3 random calls when it fails and 19 when it is successful. Using an upper bound of 12 tree attempts, a list of possible number of calls is precomputed with dynamic programming.
- Checks whether this chunk generates small trees or big trees, and filters the seed out if it’s big trees.
- From the start of tree generation, simulate the tree generation forwards. A tree attempt fails if it’s not in the same location as in pack.png, and is successful if it is in the same location as in pack.png, this should find all matching seeds but can also generate false positives which will be filtered out by later programs. In the case of the blobby tree, any tree in that area is successful*.
- On iterations where all trees we are looking for have been generated, we skip over the rest of the generation as fast as we can to the waterfall generation, and test whether any of the 50 waterfall attempts generate a waterfall in the correct location.
- Runs the LCG (RNG) backwards to the start of population to get the chunk seed**.

* This may be a bit of a risky assumption. Trees can’t generate directly next to another tree that has already generated there, but may generate 2 blocks away, and seed-reversal-gpu may miss a small proportion of seeds, where the first tree in that area is supposed to fail (the blobby tree area is quite hilly so trees might fail there in some locations), but seed-reversal-gpu assumes the attempt to be successful. This may be something we have to come back to later if we don’t find the seed, one workaround is to try the case where the first tree in that area fails but the second one is successful, which shouldn’t produce nearly as many seeds to try, and only affects x3 to x12_blobby.

** This step may fail if there was a dungeon in the chunk, which there is a 2.5% chance of.

Seed-tester-native

https://github.com/pack-png-mods/seed-tester-native

Seed-tester-native takes the output list from seed-reversal-gpu and performs some slower checks (and checks not well-suited for a GPU) on the CPU to filter out 86% of its output. The checks it carries out are as follows:

- It double checks everything seed-reversal-gpu already checked (just to be sure).
- More accurate testing for tree spawning - rejects seeds where trees are attempted to spawn in wrong locations that would succeed in the pack.png image.
- A small number of tree leaves are tested (the ones we can make out for sure).

Seed-reversal-merged

https://github.com/pack-png-mods/seed-reversal-merged

For x0 to x4, seed-reversal-gpu generates too many seeds to reasonably store on an SSD or transfer between computers, so seed-reversal-merged is basically seed-reversal-gpu and seed-tester-native merged into one program, to remove the need for the intermediate file.

Reversing world seeds (legacy section)

Converting chunk seeds to world seeds

https://github.com/pack-png-mods/chunk-to-world-seed

This program takes in a list of chunk seeds and a guess on the waterfall coordinates and outputs a list of world seeds with those chunk seeds at those coordinates. The coordinate guesses are exact in the Z axis and come from the filtered down list from the dirt anomaly in the X axis.
That's all you need to know about this section, but if you're in the mood for some maths, read the rest of it!

Recall that the chunk seed hash is implemented as

```
rand.setSeed(worldSeed);
long a = rand.nextLong() / 2 * 2 + 1;
long b = rand.nextLong() / 2 * 2 + 1;
rand.setSeed((chunkX * a + chunkZ * b) ^ worldSeed);
```

Our hope is that given a guess for x and z and the value rand is set to, we can easily recover the value of worldSeed. The first step in looking for such a method is to examine how nextLong() is implemented. We find

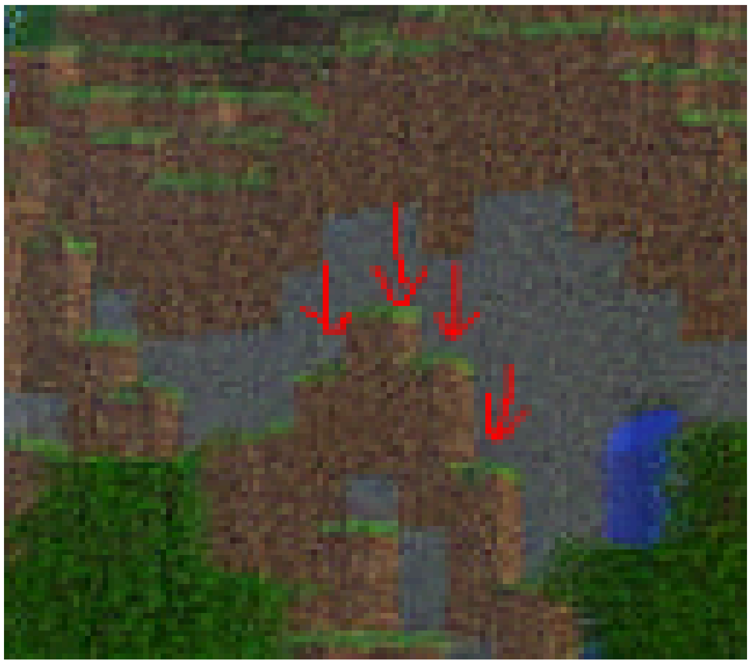
```
public long nextLong() {
   return ((long)next(32) << 32) + next(32);
}
```
Where `next(32)` runs
```
(seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1)
```
Then returns
```
(int)(seed >>> (48 - bits)). //bits = 32 in this case
```
A close examination of `nextLong()` reveals a flaw. Though `seed` is a 48 bit number, the bottom `n` bits of `nextLong()` depend only on the bottom `n + 16` bits of the seed for some `n`. This dependency extends for the most part to the bottom `n` bits of the sum `chunkX * a + chunkZ * b`. To give a concrete example, if you only know the bottom 18 bits of `seed`, then you can work out the bottom 2 bits of the chunk seed.

Now that we are aware of this flaw in `nextLong()` and the sum within the hash, we can engage ourselves with the following thought experiment. Imagine that along with `x`, `z` and the chunk seed, we knew the bottommost 16 bits of the `worldSeed;` That is, `worldSeed` $= d * 2^{16} + c$, where we know `c` and we don't know `d`, and `c` is 16-bit. Then, as we know the value of the chunk seed
```
(chunkX * a + chunkZ * b) ^ worldSeed,
```
we could undo the xor by `worldSeed` and recover the bottommost 16 bits of the sum
```
chunkX * a + chunkZ * b,
```
which we know will only depend on the bottommost `16+16 = 32` bits of the world seed.
If we were to plug in $d * 2^{16} + c$ for `worldSeed` in the `nextLong()` formula and compare the result to the value we recovered by undoing the xor, we would recover an expression of the form $m*d = e \pmod{2^{16}}$, where `m` is a subexpression only depending on `x` and `z`, and `e` is also a subexpression made of variables we already know. If we solve this equation for `d`, we will obtain an expression for the bottommost 16 bits of `d`!
This equation is solvable by multiplying both sides by the multiplicative inverse of `m` modulo 2^{16}, which we can calculate with knowledge of only `x` and `z`. We can even take this one step further, using our new knowledge of `worldSeed` to undo more of the xor and recover 16 more bits of `worldSeed` via the exact same process. Since `worldSeed` is 48 bits, we see that we are done. We have now achieved something wonderful, recovering the entirety of the `worldSeed` from just `x`, `z`, a chunk seed, and the bottom 16 bits of `worldSeed`.
One final detail of note before we abandon the thought experiment is that we don't actually know exactly how
```
rand.nextLong() / 2 * 2 + 1
```
will change with respect to just the output of the `nextLong()` call, as the way Java implements integer division depends on both the sign and the parity (whether it's odd/even) of the output of `nextLong()` and can result in 3 separate outcomes. Luckily, it is not much trouble to adjust our equations for `e` to account for each of the `3*3 = 9` possibilities for both `nextLong()` call's rounding.

Of course, in practice we don't know the bottom 16 bits of `worldSeed`, but this is not much of an issue. The simplest way around the obstruction is to iterate through all $9 * 2^{16}$ = 589,824 possible combinations of bottom 16 bits and ways `nextLong()` may have rounded, recover the world seed in that case, and check if the upper 16 bits of the chunk seed match our desired chunk seed. Since the actual reversal process is so cheap and this method does $9 * 2^{16}$ reversals per chunk seed, so long as $9 * 2^{16}$ * number of chunk seeds is less than the number of world seeds * chunks to check, this will be a significant boost in speed.

Terrain filter

https://github.com/hube12/TerrainAlphaTemp



We first check for valid biomes which are forest, seasonal forest, plains and shrubland. Then we check for those 4 blocks' heights which are 77,78,77,75 from left to right. They are at z=12 to 15 in the chunk -3 (z axis) so z=-36 to -33. The x is entirely determined by the waterfall choice made before (-1 to account the block difference).

Manual testing

The terrain filter outputs few enough seeds to test manually (approximately 1 for every 78 million), so we go through each seed it outputs and check it manually with the seed selector mod.

World Seed Filtering

Using statistics to find the x-coordinate

Filtering world seeds via sand