By Viktor40

!! This document is very WIP, it can still contain mistakes, also I'm not perfect nor 8000 IQ megabrain so even the finis hed document might contain mistakes !!

Ticks:

Basic tick loop:

---------- runServer() ----------

`net.minecraft.server.MinecraftServer.runServer`

This is the main server loop.

The main loop:

1) Initialise the loop
2) If the server is running, check if the server can keep up (i.e. sub 50 mspt)
   If not, log a warning
3) Create a new tick monitor
4) Start the tick
5) call `net.minecraft.server.MinecraftServer.tick`
6) Wait for the next tick
7) End the tick and stop the tick monitor

If an exception occurs this loop will shut down the server and try to create and save a crash report.
This will also create the main game thread.

---------- MinecraftServer.tick() ----------

`net.minecraft.server.MinecraftServer.tick`

This method is called in runServer()

1) Basic tick initialisation:

Increase the number of ticks by one
Get the time of the start of the tick.

2) Calling `net.minecraft.server.MinecraftServer.tickWorlds`

3) Some player managing
4) Autosave if the ticks is divisible by 6000 (`this.ticks % 6000 == 0`)
5) Update snooper if ticks is divisible by 6000
6) Tallying (calculate the length of the tick in ms)

---------- tickWorlds() ----------

`net.minecraft.server.MinecraftServer.tickWorlds`
This method is called in MinecraftServer.tick()

1) Tick the command function manager
2) Sync world time with other dimensions each 20 ticks (1 second)
3) call `net.minecraft.server.ServerWorld.tick` which is the main in game loop
4) Tick networkIO
5) Update player latency
6) Server GUI Refresh

---------- ServerWorld.tick() ----------

`net.minecraft.server.ServerWorld.tick`
This method is called in ServerWorld.tickWorlds()

1) World border

2) Weather

- rain
- thunder
- player sleeping

3) chunk stuff

- purge unloaded chunks
- does TACS stuff, i'm not gonna look i'm sorry
- Ticks chunks:
  > Update chunk load levels
  > Natural mob spawning
  > Mobspawning by ticking spawners
  > Random ticks

* Thunder: spawn skeleton horses and do lightning bolts
* Ice and snow: do snowing and ice forming
* Randomticks blocks in the chunk

> Player movement

- Unload Chunks (ticks TACS)
  > write POI
  > Unload the chunks
  > Initialise the chunk cache

4) block ticks

> Cleaning out the tick cache to differentiate between old ticks and ticks that still need to happen

> Schedule a block tick in case it's valid

5) fluid ticks

> Cleaning out the tick cache to differentiate between old ticks and ticks that still need to happen

> Schedule a fluid tick in case it's valid

6) raids

> Check if ongoing

> Do raid stuff

7) block events

> Call the synchedBlockEvent hashmap and execute the block events stored in it

8) entities

- If there is an ongoing dragon fight: tick the dragon
- check despawn
- tick entities
- remove entities from the world
- actually populate the spawns in the world / loads them into the world, actually different to the spawning in the chunk ticks

9) block entities

World border:

`net.minecraft.world.border.WorldBorder.tick` is called in the game loop. The only thing that will happen during this “ticking” of the world border is checking if the world border needs to change in size, and executing that.

Weather:

Weather cycle:

In the weather section of the tick function there are 6 variable allocations.
The first one is a variable you get from a function that decides if it’s raining or not from rain gradient (the sky colour, but more later on this). The other variables are clearWeatherTime, rainTime, thunderTime, isRaining and isThundering from the world properties.

The first part of the code is an if block that checks if `clearWeatherTime` is bigger than 0. This part is basically a hackfix to make it so that the weather will stay clear for a certain amount of time when the `/weather clear [time]` command is used. In this block `clearWeatherTime` will count down. After this `thunderTime` and `rainTime` will be set to 1 if it’s not thundering or raining respectively. If it is thundering or raining they will be set to 0. After this `thundering` and `raining` will be set to false in the properties.
What this accomplishes will be that once `clearWeatherTime` has reached 0, it will start raining and thundering the next tick.

If `clearWeatherTime` isn’t bigger than 0, `thunderTime` and `rainTime` will be checked. If `thunderTime` is bigger than 0 we’ll check if `thunderTime` - 1 is equal to 0. If that is the case, then the value indicating if it’s thundering (`thundering`) will be reversed. This means if it was thundering, it will stop thundering and if it wasn’t thundering it will start thundering.
If however `thunderTime` isn’t bigger than 0, a new value is set for thunder time. This value is decided by: `thunderTime = thundering ? this.random.nextInt(12000) + 3600 : this.random.nextInt(168000) + 12000;`

In a more readable format, this will mean that:
If it is thundering, the new `thunderTime` will be between 3 600 and 15 600.
If it is not thundering, the new `thunderTime` will be between 12 000 and 180 000.
Which value somewhere in this range it becomes is determined by a random factor.

The same process happens for `rainTime`, though the new value for `rainTime` is determined differently: `rainTime = propertiesRaining ? this.random.nextInt(12000) + 12000 : this.random.nextInt(168000) + 12000;`

In a more readable format, this will mean that:
If it is thundering, the new `rainTime` will be between 12 000 and 24 000.
If it is not thundering, the new `rainTime` will be between 12 000 and 180 000.
Which value somewhere in this range it becomes is determined by a random factor.
After this the values that were changed will be reallocated to the corresponding fields.

So now for the explanation of how this actually works. The way in which this works is a little confusing `clearWeatherTime` isn't used to determine how long the weather will stay clear outside of commands. Instead what happens is:
The game starts and `clearWeatherTime`, `rainTime` and `thunderTime` are set to 0. `raining` and `thundering` are set to false. Now, since everything is set to 0 it will immediately go to the part where it sets the timers, using the formulas we've just seen. In this case `rainTime` and `thunderTime` will be used to indicate how long it will **not** be raining or thundering. Both timers will count down, and only when they have reached 0 it will start raining or thundering, depending on which timer ran out. It will start this weather by flipping the boolean as we've seen before. After this a new time will be set for `rainTime` or `thunderTime`. This time, they'll be used to indicate how long it will be raining or thundering. By constantly flipping the time `rainTime` and `thunderTime` represent, there's no need to use `clearWeatherTime`. Thus we only have a value for this if we want to force the game to keep the weather clear for a certain amount of time no matter what.

Rain and thunder gradients:

A whole separate part of the weather are rain and thunder gradients. These gradients are used to determine the `ambientLightLevel` and the colour of the sky. As you know the sky darkens when it starts to rain or thunder and brightens again when the weather becomes clear. Well these gradients will take care of that.

When the weather is clear, both gradients will be 0. The value for these gradients will always be between 0 and 1 inclusive. If it gets higher than 1 it will be set back to 1. If it gets lower than 0 it will be set back to 0. Each tick the following happens. If it is raining, each tick we'll add 0.01 to the rain gradient. If it's not raining we'll subtract 0.01. This way when the sky will take 100 ticks or 5 seconds to change colour completely.

Now there is a `isRaining` and `isThundering` method, boh will determine if it's raining or not via the gradient. These functions are used to determine the weather outside of the tick loop.

Rain will only begin to fall when the gradient is bigger than 0.2 and will only stop falling when it's smaller than 0.2 again. In this case we're speaking visually of course. Internally the properties are already set when the gradients start changing.

Sleeping:

Next in this part of the tick sleeping happens. First the time of day will be set to morning if a player has just finished sleeping. It will also reset the weather if the weather cycle is enabled. Resetting the weather set `rainTime` and `thunderTime` to 0 and `raining` and `thundering` to false.

Daylight Cycle:

Curiously, the time of day will also change within the weather section. In this part the world time will be increased by one. If daylight cycle is enabled, the time of day will also change.

What about lightning strikes?

Well lightning strikes don't happen in this part of the tick but rather in the chunk part, which is the next part we'll cover.

Chunks:

Purging:

IDK when i want to look at this but not now tbh.

Chunk ticks:

Block and Fluid ticks:

Raids:

Block Events:

Order of processing:

In the block event phase, the game calls `processSyncedBlockEvents()` from `net.minecraft.server.ServerWorld`. All block events are stored in the synchedBlockEventQueue. This queue is an ObjectLinkedOpenHashSet. This is a type specific linked hash set which uses a hash table to represent a set. For more information about has sets see Earthcomputers Video. This is a linked hash set which means a list is stored as well as a hash set. This list is used for iteration.

```
public int hashCode() {
   int i = this.pos.hashCode();
   i = 31 * i + this.block.hashCode();
   i = 31 * i + this.type;
   i = 31 * i + this.data;
   return i;
}
```

this.pos.hashCode():

```
public int hashCode() {
   return (this.getY() + this.getZ() * 31) * 31 + this.getX();}
```

this.block.hashCode():

```
public int hashCode() {
   int i = this.self.hashCode();
   i = 31 * i + this.other.hashCode();
   i = 31 * i + this.facing.hashCode();
   return i;}
```

this.self.hashCode(), this.other.hashCode() and this.facing.hashCode()
Use the standard hashCode() method from the Object class in java. `self` and `other` are `net.minecraft.block.BlockState` objects. `facing` is a `net.minecraft.util.math.Direction` object.

This means that a block events hashCode depends on:

- It's position (x, y and z)
- the type of block. All different attributes of the block are taken into account. But things like it's position are not going to affect this particular part of the hash code. This means that this part should be the same for all blocks with the same attributes and thus should be the same for 2 exact copies of a block in different positions.
- The type of block event (see next section)
- the data of the block event (see next section for how data is determined)

`processSynchedBlockEvents` will go through the `syncedBlockEventQueue` until it's empty. First there will be a check. Here the game will get the `BlockState` of the block in the position where the `BlockEvent` is called. If this `BlockState` corresponds to the block state of the `BlockEvent` then processing will continue. If not the code will go to the next object in the queue. This can happen when for example the block is broken or has changed state already.

After this the `onSynchedBlockEvents` method will be called. After this there are 2 possibilities. It's either a block entity or a normal block. There is a difference between those two in what happens. More about this will be discussed later

Types, data and specifics (addSynchedBlockEvent):

The order in which block events happen is both locational and directional since it is stored in a hashset.

Things that call:
`net.minecraft.server.world.serverworld.addSyncedBlockEvent`

- Bell Block
  Type = 1
  data = `direction.getId()`, where direction encomasses the side on which the bell block was hit.
  A new block event is scheduled when a bell block is hit.

- Note Block
  Type = 0
  data = 0
  A block event is scheduled when a note is played

- Piston Block
  Extending
  Type = 0
  data = the id of the direction the piston is facing

  Retracting
  Type = 1
  data = the id of the direction the piston is facing

  Moving
  Type = 2
  data = the id of the direction the piston is facing

  A piston block creates block events when it starts extending or retracting and during it's movement.

- Chest Block
  Type 1 = 1
  data = the amount of players viewing the inventory
  When opening or closing the inventory

- End Gateway Block
  Type 1 = 1
  data = 0
  When it starts the teleport cooldown after teleporting an entity

- Ender Chest Block
  Type = 1
  data = the amount of players viewing the inventory
  When the enderchest gets ticked (once every second)
  When opening or closing the ender chest

- Mob Spawner Block Entity
  Type = 1

data = 0
When Updating spawns

- Shulker Box Block Entity
  Type = 1
  data = the amount of players viewing the inventory
  When opening or closing the shulker box

What does a block event do? (onSynchedBlockEvent)

So now we know when a block event gets scheduled, how it's hash works, what actions schedule block events etc. But we don't yet know what it does, why block events are used etc.

First the `onSynchedBlockEvents` method from `AbstractBlock` gets called. This will then call a different method depending on if the block is a normal block or a block entity.

Block Entities:

Block entities is the biggest category. Here the `onSynchedBlockEvents` method from `BlockWithEntity` gets called. First will super the method from `AbstractBlock`. This is a deprecated method and will always return false, thus won't do anything. After this it will check if the bloc entity isn't null. Now `blockEntity.onSyncedBlockEvent(type, data);` gets called. This will first call the block specific method and then super some methods.

Bell Block:

The bell block will ignore the block event if the type isn't 1 (return false). If the type is 1 the following happens:
First of all the bell memories get notified. This mostly does some variable allocations
Secondly, the amount of ticks the bell is resonating is set to 0.
The side on which the bell got hit is recorded.
The ringing time gets set to 0 and ringing is set to true.
then true gets returned.

Chest Block:

If the type is one, it will set viewerCount = data and return true. Else it will return false.

End Gateway Block:

If the type is one, it will set the teleportCooldown = 40 and return true. Else it will return false.

Ender Chest Block:

If the type is one, it will set viewerCount = data and return true. Else it will return false.

Mob Spawner Block:

If the type is 1 and it's client side, spawn delay = min spawn delay and true is returned. Else false is returned.

Shulker Box:

if type isn't 1, return false.
If data = 0 close the shulker box and return true
if data = 1 open the shulker box and return true

Blocks:

For normal blocks it's a little simpler. Here the block specific method gets called immediately. After this super methods get called.

Comparator Block:

All the code about block events in comparator block classes doesn't do anything. It is most likely leftover or something. There's also no way to have a block event on the position of a comparator execute so yeah ...

Note Block:

play the note, display the particles and make the sound in game, then return true

Piston Block:

I'm going to stay away from pistons for a bit.

Entities:

Mobs:

Order of processing:

Hierarchy:

1) `net.minecraft.server.world.ServerWorld.tickEntity`
2) Entity specific class with a `tick` method
    a) `net.minecraft.entity.mob.MobEntity.tick` gets supered.
    b) `net.minecraft.entity.LivingEntity.tick` gets supered
    c) `net.minecraft.entity.Entity.tick` gets supered
        i) `net.minecraft.entity.mob.MobEntity.basetick` gets called
        ii) `net.minecraft.entity.LivingEntity.basetick` gets supered
        iii) `net.minecraft.entity.Entity.basetick` gets supered

…

More detail on this will come in the next sections:

---------- ServerWorld.tickEntity ----------:

First the game looks if the entity needs to be ticked. It doesn't tick the entity if either the entity is an object of the player class or if `getChunkManager().shouldTickEntity(entity)` is false.
If either of these conditions aren't satisfied `this.checkEntityChunkPos(entity)` is called.

If the entity needs to be ticked the following happens:

- The position, pitch, yaw etc. variables get reassigned from `entity.prevYaw` to `entity.yaw`
- If the entity needs to be updated, its age gets advanced by 1.
  After this the entity will be registered in the profiler and the entity will be ticked.
- After this the entity's chunk position will be checked.
- Finally, if there are any passengers they will be ticked too.

In the following sections we'll be looking at everything that happens in a chronological order

---------- SpecificMob.tickEntity ----------:

Most mobs override the tick method from MobEntity.java. In this method the method from MobEntity gets supered.
We'll look at specific mobs after we've described the general way in which mobs work.

---------- MobEntity.tick() ----------:

This overrides the tick method from LivingEntity.

- First the tick method from LivingEntity gets supered
- After this it checks if the instance is client or server
  If it's server it will update the leach if the mob has one
  it will also update controls every 5 ticks: basically enabling or disabling whether the entity can take control of the mob

---------- LivingEntity.tick() ----------:

This overrides the tick method from Entity.
A lot of this stuff doesn't apply to mobs though, but only to players

- First the tick method from Entity gets suppered
- After this `tickActiveItemStack()` is called. This basically updates the item someone is using

- After that it updates the leaning pitch (swimming or not)
- The following only happens if the world is an instance of server
    a. update despawn timers of stuck arrows
    b. update despawn timers of stuck stingers
    c. something with armour slots
    d. update the damage tracker
    e. update if the mob should be glowing
    f. update sleeping (waking up or not)
- Movement (call tickMovement -> see later)
- hand animation and position
- yaw and bobbing
- head turning
- range checks (basically makes sure tilts and angles are between -180° and 180°)

---------- Entity.tick() ----------:

First it checks if the entity is glowing. If it is it sets a flag for that.
After this the baseTick() method gets called.

---------- MobEntity.baseTick() ----------:

This overrides the baseTick method from LivingEntity.

- The super method gets called
- Does sound stuff

---------- LivingEntity.baseTick() ----------:

This overrides the baseTick method from Entity

- update hand swinging variables (old -> new)
- sets bed position if needed
- displays soul speed effect if needed
- The super method gets called
- Damage the entity if it’s inside a wall
- Damage an entity if it’s outside of the world border
- Extinguish fire if needed
- If the entity is under water it updates the air bubbles and damages it if needed
- Kicks the entity from a vehicle if it can’t be ridden on water
- Updating of hurt timer and regen timer
- Checks if the entity has died
- Status effects
- updating of movement and looking variables (old - > new)

---------- Entity.baseTick() ----------:

This is the upper most baseTick method

- Stop riding vehicles if needed
- change riding cooldown
- update some movement variables
- tick nether portals
- spawn particles
- Update water stuff: swimming etc
- extinguish fire
- set on fire if in lava
- kill the entity if it's y value is lower than -64
- set a flag for the amount of ticks it needs to stay on fire

---------- tickMovement() ----------:

This is called within LivingEntity.tick()
If it's a passive animal the first method that will be called is the method from `net.minecraft.entity.passive.PassiveEntity`
If it's not there's a mob specific method. In the next sections i will go over all the tickMovement methods from PassiveEntity.tickMovement() all the way up to LivingEntity.tickMovement(). For non passive entities the specific methods will be adressed in the sections about that mob.

---------- PassiveEntity.tickMovement() ----------:

This overrides the tick method from MobEntity.

- call the super method
- call the super method
- Anything in here only happens client side
  The following happens only when happyTicks is greater than 0.
  HappyTicks is set to 40 when a baby grows up and is used for particles.
  a. If happy ticks is still greater than 0
  b. if happyTicks % 4 = 0: emit particles
  c. remove 1 from the happyTicks value
- This only happens on the server side
  a. sets the breeding age

---------- MobEntity.tickMovement() ----------:

This overrides the tick method from LivingEntity.

- super the method from LivingEntity
- this happens only on the server side:
  if the mob can pick up loot and
  if the mob is alive and
  if the mob is not dead and
  if gamerule mob griefing is true:

the mob will try to pick up stuff from the ground

---------- LivingEntity.tickMovement() ----------:

Most of the movement stuff happens in this method.

- tick down jumpingCooldown
- body tracking
- change jaw, pitch, body elements, head etc
- change velocity
- If the mob is immoble set all speeds to 0
- if the mob can move voluntarily:
  call tickNewAI (see next section)
- jumping
  a. look at the height of fluids
  b. swimming upwards
  c. jumping
- travel with the different speeds set in tickNewAI
- pushing
  a. tick down riptideTicks and tick riptide
  b. do cramming
- drowning

---------- MobEntity.tickNewAI() ----------:

this is being called in LivingEntity.tickMovement() and handles all of the AI stuff. I'm going to seperate this in sections based on the profiler loggings.

1) add to the despawn counter
2) sensing
3) targetSelector
4) goalSelector
5) navigation
6) mob tick
7) controls
8) move
9) look
10) jump

Sensing:

- clear the visibility cache
  This contains visible and invisible entities

targetSelector:

here targetSelector gets ticked, which is a GoalSelector object

what happens in the tick() method:

- remove unnecessary goals
- update goals
- tick each goal

goalSelector:

call this.goalSelector which is an GoalSelector object.
what happens in the tick() method:

- remove unnecessary goals
- update goals
- tick each goal

navigation:

call this.navigation which is an EntityNavigation object.

- recalculate the path if needed
- return if the mob is idle
- if the mob is at a valid position, continue following the path
- if the path is finished, go to the next one

mob tick:

This is a method that's different for each mob and will be addressed for each mob specifically.

move:

calls MoveControl.tick()

look:

looking

jump:

jumping

Block Entities:

Mobs

Mob Spawning:

---------- ServerChunkManager.tickChunks() ----------:

Mob spawning happens early in the tick inside ServerChunkManager.tickChunks().
The important things for mob spawning that happens are:

```
boolean timeModulus = levelProperties.getTime() % 400L == 0L;
```

This timeModulus will only be true when the world time divided by 400 is an integer. (i.e. the world time is divisible by 400). This is used for the passive mob spawning cycle.

Now there is a loop going over all loaded chunks.
In this loop, for each chunk the game checks if there's a non spectator player inside a 128 radius of the chunk. If this is not true, mob spawning won't even happen in this chunk. The method will be terminated with a return.

If this is the case, the local difficulty will increase.
After this mobs will spawn if the gamerule doMobSpawning is true and if the chunks are within the world border. `SpawnHelper.spawn()` is called.

---------- SpawnHelper.spawn() ----------:

In this method, the game will iterate over all spawn groups. The different spawn groups can be found in SpawnGroup.java and can be found in the next section.
It checks for a few conditions before proceeding with mobspawning. Spawning will terminate if either of the following conditions is true.

- The spaw group is animal (CREATURE) and the timeModulus is false time not divisable by 400
- The number of loaded mobs exceeds the mobcap

After these checks, SpawnHelper.SpawnEntitiesInChunks() is called.

Spawn Groups:

SpawnGroup(String name, int spawnCap, boolean peaceful, boolean isAnimal, int despawnRange)

```
MONSTER("monster", 70, false, false, 128),
CREATURE("creature", 10, true, true, 128),
AMBIENT("ambient", 15, true, false, 128),
WATER_CREATURE("water_creature", 5, true, false, 128),
```

```
WATER_AMBIENT("water_ambient", 20, true, false, 64),
MISC("misc", -1, true, true, 128);
```

The first value is the name of the group.
The second value is the mobcap.
The third value indicates whether the mob can spawn in peacful mode.
The fourth value indicates if the mob is an animal
The fifth value indicates the size of the despawn sphere.

- Monster -> Hostile mobds
- Creature -> animals
- Ambient -> bats
- Water_Creature -> squid, dolphins etc.
- Water_Ambient -> fish
- Misc -> A special group only used for mob spawning

There are some noteable things to notice in here:
- Every spawn group has a despawn sphere of size 128 except for fish
- The Misc group doesn't have a mobcap (doesn't get used for natural spawning)

---------- SpawnHelper.spawnEntitiesInChunks() ----------:

There are multiple methods with the same name. They are called in an order and I will go over all of them in the same section in the order they get called.

First an initial spawn position is chosen inside the chunk. This happens in the following manner:
- X position: random position within the chunk
- Z position: random position within the chunk
- Y position: The Y position is decided by the heightmap. See section on heigtmap.

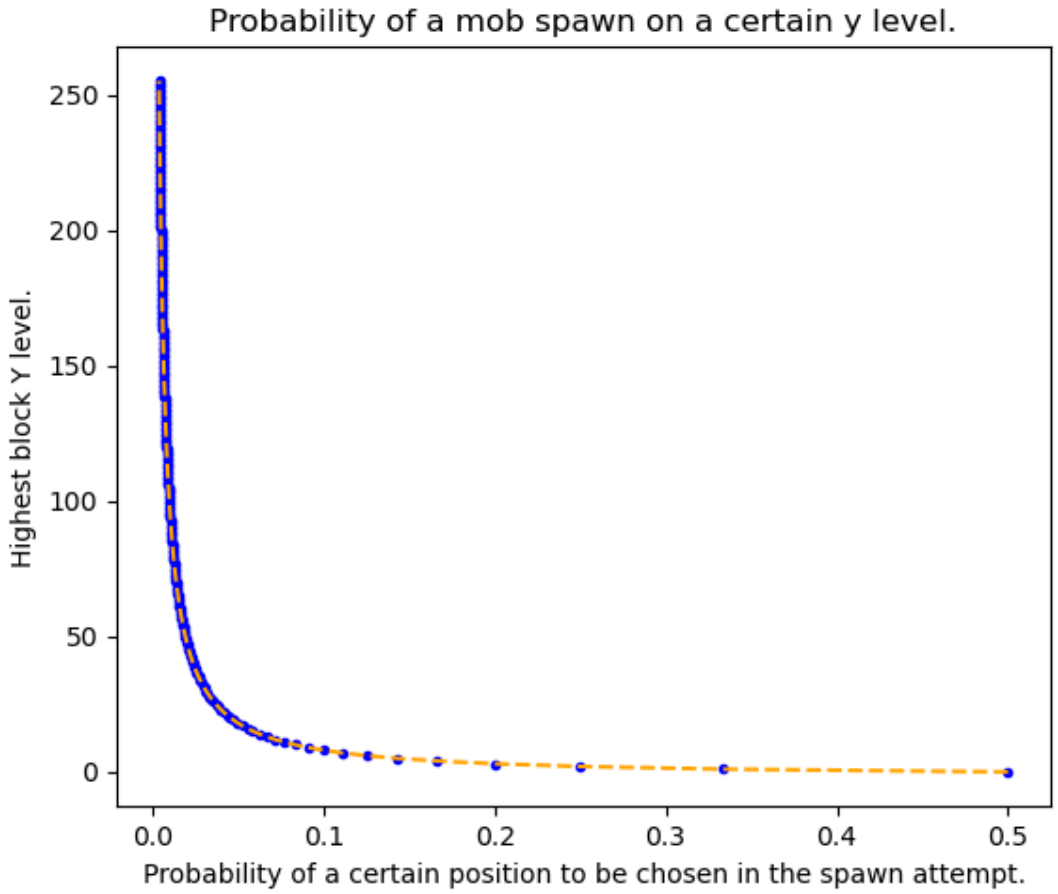
Now we have chosen a random position to spawn the mob inside the chunks. If this position is lower than y = 1 then the spawn attempt will be terminated.

Heightmap in regards to mob spawning:

When the heightmap generates it goes over each block in the chunk. It searches for the highest subchunk with blocks inside. When it has found this, the game will start from the top of the subchunk and work it's way down, until it finds a block that isn't air. When it has found this block, the heightmap for this xz position will be the y position of the highest block +1.

When the y position for mob spawning is decided the following happens:
A random number inside the interval [0, heightmap +1] = [0, highest block +2] both inclusive, gets chosen. This position is then passed as the y position for the spawn attempt.
The probability of a certain position being chosen is:
P(spawn) = 1 / (Highest block + 2) = 1 / (highest possible spawn position + 1)

Examples (Y is the highest block in the 1x1 block column):
Y = 0: P(spawn) = 1 / (0 + 2) = 1 / 2 = 0.5
Y = 2: P(spawn) = 1 / (2 + 2) = 1 / 4 = 0.25
Y = 60: P(spawn) = 1 / (60 + 2) = 1 / 62 = 0.0161
y = 200: P(spawn) = 1 / (200 + 2) = 1 / 202 = 0.00495
As you see the probability of a mob spawning increases rapidly the higher you get.
This is also shown in the following image:



The Villager:

---------- VilagerEntity.tick() ----------:

- super tick()
- change headRollingTimeLeft
- decay gossip

---------- VilagerEntity.mobTick() ----------:

Instantiate a raid variable which is an object from the Raid class

- tick the brain

- change levelUpTimer
-