Facultad
Informática
Universidad
Murcia

Universidad de Murcia
Facultad de Informática

GRADO EN INGENIERÍA INFORMÁTICA

La evolución de los errores de programación en los videojuegos: La curiosa situación actual de Minecraft.

Trabajo de Fin de Grado realizado por:
Adrián Muelas Gómez
{adrian.muelas@um.es}

Bajo la dirección de:
Francisco de Asis Guil Asensio
{fguil@um.es}
Eduardo José Gómez Hernández
{eduardojose.gomez@um.es}
Junio, 2023

Esta página ha sido intencionadamente dejada en blanco

Erratas

- Corregido los nombres de las variables del Código 11.

Disclaimer

Índice

Resumen

Los ***exploits*** son errores de programación (*bugs*) que el jugador puede utilizar para su propio beneficio. Hay comunidades que se dedican a buscar *exploits* en videojuegos, habitualmente para su uso en retos auto impuestos como el *speedrunning*, lo cual involucra a menudo hacer ingeniería inversa al juego para poder ver cómo funciona por debajo.

Minecraft, el videojuego más vendido de la historia y con mayor impacto cultural a día de hoy, no es una excepción. Este se caracteriza por fomentar la creatividad del jugador ofreciéndole un grado de libertad casi absoluto para modificar un mundo formado por bloques, lo que se ajusta al género *sandbox*, pero además tiene ciertos elementos de supervivencia para suplir la ausencia de objetivos fijos. Debido a esto, parte de la comunidad se dedica a intentar llevar el juego al límite para conseguir recursos y materiales usando los distintos bloques de automatización que presenta, las mecánicas del propio mundo y, a menudo, *exploits*.

A principios de 2022 la comunidad publica una serie de descubrimientos en los que han estado trabajando los últimos años, un *exploit* que permite al jugador obtener bloques que son imposibles de obtener de forma normal. Si bien no es tan crítico como en el caso de otros videojuegos, es un tipo de *exploit* único hasta la fecha y, por la serie de casualidades que da lugar al mismo, probablemente irrepetible.

El objetivo de este trabajo es entender y explicar cómo funciona este *exploit* para ver qué lo hace especial y aportar nuestro propio conocimiento para mejorar la estabilidad y eficiencia del proceso.

La investigación comienza en 2020 con el fin de obtener el objeto de un bloque indestructible. Había entonces la teoría de que sería posible conseguirlo usando bloques con gravedad. Este es un grupo de bloques que tienen la característica de que si no tienen un suelo sobre el que reposar se convierten temporalmente en una entidad que cae hasta encontrar otro bloque. Esto está hecho de forma que, si tras una comprobación y antes de crear la entidad fuéramos capaces de cambiar el bloque presente en esa posición, ésta se crearía utilizando el bloque irrompible en lugar del original.

Aunque sabemos cómo hacer que el juego coloque el bloque que queremos donde queremos, por desgracia no hay forma de hacerlo cuando queremos, pues no ocurre nada entre ambos puntos del código que podamos manipular desde dentro del juego.

Por otro lado se encontró que, al poner cierto bloque, el juego ejecuta un segundo hilo para actualizar un efecto visual. Especialmente debido a que Minecraft está dividido entre un cliente encargado del dibujado y el servidor que maneja el mundo, no debería ser un problema que haga lecturas en paralelo, pero este proceso también ocurre erróneamente en el servidor. Aunque en principio pudiera parecer que no tiene ninguna implicación adicional, el proceso de creación explicado anteriormente sería posible si se produjera una escritura en paralelo provocando lo que se conoce como *data race*, un tipo de condición de carrera que ocurre cuando más de un hilo accede a la misma localización en memoria mientras al menos uno la modifica, causando que el valor al que sea accede no sea determinista. En este caso, querríamos producirla para sustituir el bloque antes de que se cree la entidad. A pesar de que sólo hace lecturas, se encontró que, si se dieran unas condiciones específicas, existe un camino teórico para lograr escrituras.

Para poder leer un bloque, hay que acceder a la subdivisión del mundo en que se encuentra, lo que conocemos como *chunk*. Si es la primera vez que se accede a un *chunk*, éste tiene que crearse, pero este proceso está dividido en dos partes. En el primer acceso siempre ocurre la generación básica del terreno que incluye la superficie terrestre, el mar y las formaciones de cuevas y grutas. Tras eso debe ocurrir la población, que decora el mundo con diversos elementos como vegetación o estructuras, sin embargo ésta se produce en la esquina sureste del *chunk*, afectando a otros adyacentes. Es por eso que este proceso se aplaza si todos los *chunks* afectados no están cargados inicialmente y se volverá a intentar cada vez que uno de estos se cargue. Esto nos interesa porque algunos elementos producen actualizaciones, es decir, que notifican a los bloques vecinos de que ha ocurrido un cambio para que puedan reaccionar en el momento, por ejemplo, el bloque con gravedad intentaría caer al recibir una actualización. Además, de entre los que provocan actualizaciones, las fuentes de agua hacen que el

juego entre en un estado conocido como **Instant Tile Tick** en el cual, muchos de los bloques que normalmente tardan en reaccionar, lo hagan de forma instantánea y recursiva.

En este punto, la comunidad ya tiene experiencia a la hora de manipular la población y podemos repetir el proceso un número ilimitado de veces gracias a lo que se conoce como ***save state***, una técnica que consiste en aumentar el tamaño del *chunk* por encima del límite para evitar que el juego guarde los cambios, por lo que tenemos las herramientas necesarias para propagar actualizaciones asíncronas y experimentar con ellas. Sin embargo, todo esto es sólo teoría de momento, pues conseguir que esto ocurra en paralelo no es tan sencillo. Para crear el hilo necesitamos poner un bloque en el *chunk*, por lo que tiene que existir de antemano.

Finalmente la comunidad da con una complicada estrategia llamada ***Chunk Swap*** que consiste en provocar una nueva condición de carrera en la tabla *hash* que almacena los *chunks* cargados de forma que, cuando otro hilo busca el *chunk*, no sea capaz de encontrarlo y reemplace al original. El método más conocido de hacer esto a día de hoy se aprovecha de cómo está implementada la eliminación de un elemento, es decir, la descarga del *chunk*; ya que, siendo una tabla de direccionamiento abierto, se debe considerar si ha habido una colisión, por lo que, tras colocar `null` en su posición, se debe buscar si otro elemento debe que ocupar esa posición. Calculando el *hash* del *chunk* que nos interesa, podemos cargar otros *chunks* forzando que este se desplace, haciendo más lenta la búsqueda y dando tiempo a que la condición de carrera ocurra. Por otro lado, el hilo que creamos se procesa demasiado rápido, por lo que ya ha terminado cuando el juego modifica la tabla. Para solucionarlo, se aprovechan de que, dadas las condiciones, el hilo entra en un bloque `synchronized`, por lo que, al invocar múltiples hilos, estos se van bloqueando y ralentizando entre si.

Usando todas estos conocimientos, entre otros menos relevantes, finalmente se consiguió producir actualizaciones en paralelo pero, además de producir el efecto indicado por esta primera teoría, encontraron otro uso muy interesante.

Debido a la gran cantidad de datos que emplea, el juego hace uso de una paleta para reducir el espacio que ocupan los bloques en memoria. Ésta varía dependiendo del número de bloques presentes en el *chunk* y, con ella, la longitud del identificador de los bloques, lo que se conoce como tamaño de palabra. Estos son almacenados concatenados en un array de longs por lo que, si el tamaño de palabra no es potencia de dos, un bloque puede acabar cortado entre dos longs. ***Word Tearing*** consiste en provocar un *data race* durante este proceso, sobrescribiendo parte del bloque, permitiéndonos incluso obtener aquellos bloques que no están disponibles en el modo de juego por defecto. ***Generic Method*** es el nombre que se le da al uso combinado de *Word Tearing* y *Falling Block Swap* permitiéndonos conseguir los objetos de todos los bloques posibles. Esto significa que se requiere, no una, sino dos condiciones de carrera simultáneas, llevando el *exploit* a un nuevo nivel de complejidad.

Hay sin embargo diversos problemas que hemos investigado, dando nuestro propio aporte. En primer lugar, la estrategia que usaron para hacer *Chunk Swap* estaba pensada para múltiples jugadores, lo cual es un limitante importante para su uso, por lo que una de las tareas que hemos realizado ha sido diseñar una estrategia con un sólo jugador. El problema está relacionado con cómo se descarga el *chunk* en el método original. Además del jugador que crea los hilos, un jugador se posiciona de forma que el *chunk* a descargar se encuentra justo en la esquina de su rango de dibujado y otro desplazado lo suficiente como para no cargarlo. Esto es así para que cuando el segundo jugador se mueva, se descargue únicamente el que nos interesa. De lo contrario es posible que no fuera el primero en descargarse, lo cual es contraproducente porque alejaría la creación del hilo del momento de la descarga. Esto no se puede adaptar a un sólo jugador, por lo que hay que buscar otra forma de conseguir esto. Cuando ocurre el auto-guardado y, más convenientemente, al pausar el juego, todos aquellos que no deberían estar cargados son marcados para descargarse posteriormente. Aprovechando que existe un límite de descarga por ciclo de juego, podemos planificar fácilmente cuántos *chunks* están cargados para aislar el que nos interesa en un ciclo posterior.

Por otro lado analizamos cómo es posible que ocurra *Word Tearing* exactamente, pues la lectura y la escritura estaban demasiado cerca como para que la condición de carrera ocurriera con tanta frecuencia. Pensando que Java pudiera reordenar las instrucciones, hemos llegado a desensamblar la

parte del juego relevante y hemos visto que éste no era el caso. Al final determinamos que sólo es posible debido a la ejecución especulativa y fuera de orden que ocurre en el procesador, realizando la lectura y las operaciones intermedias mucho antes de que se realice la escritura.

Por último quisimos abordar la inestabilidad presente en todo el proceso, empezando porque los hilos terminaban su ejecución ocasionalmente sin lanzar ningún error. Modificando el juego ligeramente pudimos ver que ocurría una excepción que no se capturaba y, analizando la traza, determinamos que se debe a una condición de carrera imprevista al actualizar de forma asíncrona cierto tipo de bloque que almacena información extra en una entidad especial. Como debe consultarla, el hilo intenta buscarla en una estructura, pero cuando lo hace, el juego reduce el tamaño de la estructura, y acaba extrayendo `null`. Por desgracia no podemos impedir que la estructura se reduzca, así que tenemos que evitar actualizar ese tipo de bloque. Finalmente la solución fue bloquear la propagación de actualizaciones asíncronas desde el hilo principal mientras este bloque está presente.

Luego, investigamos por qué el juego parecía *crashear* en cualquier momento. Resulta que, cuando un bloque cambia dentro de un *chunk* que es visible por al menos un jugador, el juego almacena en una estructura los cambios para luego mandarlos juntos al final de cada ciclo de juego. Si mientras se está procesando se intenta añadir una entrada de un *chunk* que no había sido modificado, se lanza una excepción de modificación concurrente. Esto es fácil de solucionar garantizando que se produzcan actualizaciones en cada ciclo en cada *chunk* que modificamos en paralelo. Pero eso no es todo, curiosamente sigue *crasheando* en el mismo punto. Resulta que otra condición de carrera en aquellos que sí han sido añadidos hace que luego queden permanentemente excluidos, lo que causa una desincronización visual entre el cliente y el servidor. Sin embargo, cada 8001 ciclos, el juego actualiza todos los *chunks* visibles aunque no tenga cambios, por lo que se pueden volver a incluir en la estructura. La parte mala es que, al hacer esto, justo después vuelve a pasar por la lista de forma innecesaria pero que permite volver a provocar la condición anterior durante este ciclo. En este caso hay distintas soluciones teóricas, la más sencilla consiste en redirigir las actualizaciones fuera del rango de visión de los jugadores temporalmente. Una vez conseguimos prevenir todos los *crashes* anteriores, mantuvimos el juego haciendo *Generic Method* por más de 48 horas sin problemas, por lo que creemos que lo hemos estabilizado al menos a un nivel aceptable, aunque siempre cabe la posibilidad de que hayan otras casos extremos que no hayamos podido detectar.

Cabe mencionar que durante la investigación hemos creado varias herramientas que nos permiten visualizar distintos elementos del juego y que, además de lo que hemos explicado, también hemos analizado otros *crashes* menos relevantes y posibles optimizaciones que hemos decidido excluir del documento.

Extended Abstract

Exploits are programming errors (bugs) that the player can use for his own benefit. There are communities dedicated to finding exploits in video games, usually for use in self-imposed challenges such as speedrunning, which will often involve reverse engineering the game in order to see how it works underneath.

Minecraft, the best-selling video game in history and with the greatest cultural impact today, is no exception. It is characterized by encouraging the player's creativity by offering them an almost absolute degree of freedom to modify a world formed by blocks, which is in line with the genre of sandbox, but it also has certain survival elements to make up for the absence of fixed objectives. Because of this, part of the community is dedicated to trying to push the game to its limits in order to obtain resources and materials using the various automation blocks it presents, the mechanics of the world itself and, often, exploits.

At the beginning of 2022 the community publishes a series of discoveries they have been working on for the last few years, an exploit that allows the player to obtain blocks that are impossible to obtain in the normal way. Although it is not as critical as in the case of other video games, it is a unique type of exploit to date and, due to the series of coincidences that give rise to it, probably unrepeatable.

The aim of this work is to understand and explain how this exploit works to see what makes it special and to contribute our own knowledge to improve the stability and efficiency of the process.

The research starts in 2020 in order to obtain the object of an indestructible block. There was then a theory that it would be possible to achieve this using blocks with gravity. This is a group of blocks that have the characteristic that if they have no ground to rest on, they temporarily become a falling entity until they find another block. This is done in such a way that, if after a check and before creating the entity we were able to change the block present in that position, it would be created using the unbreakable block instead of the original one.

Although we know how to make the game place the block we want where we want, unfortunately there is no way to do it when we want, because nothing happens between the two points of the code that we can manipulate from within the game.

On the other hand, it was found that, when putting a certain block, the game runs a second thread to perform a visual effect. Especially since Minecraft is split between a client in charge of drawing and the server that manages the world, it should not be a problem that it does parallel reads, but this process also happens erroneously on the server. Although it might at first appear to have no additional implications, the creation process explained above would be possible if a parallel write were to occur causing what is known as a data race, a type of race condition that occurs when more than one thread accesses the same location in me-mory while at least one modifies it, causing the value accessed to be nondeterministic. In this case, we would want to produce it to replace the block before the entity is created. Although it only does reads, it was found that, given specific conditions, there is a theoretical way to achieve writes.

In order to read a block, the subdivision of the world in which it is located, known as a chunk, must be accessed. If it is the first time a chunk is accessed, it has to be created, but this process is divided into two parts. In the first access always occurs the basic generation of the terrain including the land surface, the sea and the cave and grotto formations. After that should occur the population, which decorates the world with various elements such as vegetation or structures, however this occurs in the southeast corner of the chunk, affecting other adjacent chunks. That is why this process is postponed if all the affected chunks are not initially loaded and will be retried every time one of them is loaded. This is of interest to us because some elements produce updates, i.e., they notify neighboring blocks that a change has occurred so that they can react on the fly, e.g., the block with gravity will attempt to drop upon receiving an update. In addition, among those that cause updates, water fountains cause the game to enter a state known as **Instant Tile Tick** in which many of the blocks that normally take a long time to react, react instantly and recursively.

At this point, the community already has experience in manipulating the population and we can repeat the process an unlimited number of times thanks to what is known as save state, a technique that consists of increasing the chunk size over the limit to prevent the game from saving the changes, so we have the necessary tools to propagate synchronous updates and experiment with them. However, all this is just theory at the moment, as making this happen in parallel is not so easy. To create the thread we need to put a block in thechunk, so it has to exist beforehand.

Finally the community comes up with a complicated strategy called **Chunk Swap** that consists of causing a new race condition in the hash table that stores the loaded chunks so that, when another thread searches for the chunk, it is not able to find it and replaces the original one. The best known method of doing this today takes advantage of how the deletion of an element is implemented, that is, the unloading of the chunk; since, being an open-addressing table, it must consider if there has been a collision, so, after placing `null` in its position, it must be searched if another element must occupy that position. By calculating the hash of the chunk we are interested in, we can load other chunks forcing it to move, slowing down the search and giving time for the race condition to occur. On the other hand, the thread we create is processed too quickly, so it is already finished when the game modifies the table. To solve this, we take advantage of the fact that, given the conditions, the thread enters a `synchronized` block, so that, when multiple threads are invoked, they block and slow each other down.

Using all this knowledge, among other less relevant ones, they finally managed to produce parallel updates but, in addition to producing the effect indicated by this first theory, they found another very interesting use.

Due to the large amount of data it uses, the game makes use of a palette to reduce the space occupied by the blocks in memory. This varies depending on the number of blocks present and, with it, the length of the block identifier, known as the word size. These are stored concatenated in an array of longs so that, if the word size is not a power of two, a block may end up cut between two longs. **Word Tearing** consists in causing a datarace during this process, overwriting part of the block, even allowing us to obtain those blocks that are not available in the default game mode. **Generic Method** is the name given to the combined use of Word Tearing and Falling Block Swap allowing us to get the objects of all possible blocks. This means that not one, but two simultaneous run conditions are required, taking the exploit to a new level of complexity.

However, there are several problems that we have investigated, giving our own contribution. First, the strategy they used to make Chunk Swap was intended for multiple players, which is a major limitation for its use, so one of the tasks we have done has been to design a single-player strategy. The problem is related to how the chunk is unloaded in the original method. In addition to the player creating the threads, one player is positioned so that the chunk to be unloaded is right in the corner of his drawing range and another player is offset enough not to load it. This is so that when the second player moves, only the one we are interested in is unloaded. Otherwise it is possible that it would not be the first to unload, which is counterproductive because it would move the creation of the thread away from the time of unloading. This cannot be adapted to a single player, so another way to achieve this must be found. When auto-saving occurs and, more conveniently, when pausing the game, all those that should not be loaded are marked for later unloading. Taking advantage of the fact that there is a limit of unloading per game cycle, we can easily plan how many chunks are loaded to isolate the one we are interested in in a later cycle.

On the other hand, we analyzed exactly how Word Tearing could occur, since the read and write were too close for the race condition to occur so frequently. Thinking that Java might reorder the instructions, we went so far as to disassemble the relevant part of the game and found that this was not the case. In the end we determined that it was only possible because of the speculative, out-of-order execution that occurs in the processor, performing read and intermediate operations long before the write is performed.

Finally, we wanted to address the instability present in the whole process, starting with the fact that the threads occasionally terminated their execution without throwing any error. By modifying

the game slightly, we were able to see that an exception occurred that was not caught and, by analyzing the trace, we determined that it was due to an unforeseen race condition when synchronously updating a certain type of block that stores extra information in a special entity. Since it must query it, the thread tries to look it up in a structure, but when it does, the game reduces the size of the structure, and ends up extracting `null`. Unfortunately we can't stop the structure from shrinking, so we have to avoid updating that kind of block. Finally the solution was to block the propagation of synchronous updates from the main thread while this block is present.

Next, we investigated why the game seemed to crash at any time. It turns out that, when a block changes within a chunk that is visible to at least one player, the game stores the changes in a structure and then sends them together at the end of each game cycle. If, while processing, an attempt is made to add an entry from a chunk that has not been modified, a concurrent modification exception is thrown. This is easy to fix by ensuring that updates occur every cycle on every chunk we modify in parallel. But that doesn't work, curiously it still crashes at the same point. It turns out that another race condition on those that have only been added causes them to be permanently excluded afterwards, which causes a visual desynchronization between the client and the server. However, every 8001 cycles, the game updates all visible chunks even if it has no changes, so they can be re-included in the structure. The bad part is that, when doing this, right after it goes through the list again in an unnecessary way but it allows to cause the previous condition again during this cycle. In this case, there are several theoretical solutions, the simplest of which is to redirect the updates out of the players' range of view temporarily. Once we managed to prevent all the previous crashes, we kept the game running Generic Method for more than 48 hours without problems, so we believe that we have stabilized it at least at an acceptable level, although there is always the possibility that there are other extreme cases that we have not been able to detect.

It is worth mentioning that during the research we have created several tools that allow us to visualize different elements of the game and that, in addition to what we have explained, we have also analyzed other less relevant crashes and possible optimizations that we have decided to exclude from the paper.

1 Introducción

El objetivo de este trabajo es observar cómo han evolucionado los **errores de programación**, comúnmente conocidos por el término inglés ***bug***, así como el impacto de estos en los jugadores. Para ello necesitamos hablar primero de la terminología que ha ido apareciendo en las comunidades para diferenciar distintos aspectos conforme se ha ido teniendo un mayor conocimiento, aunque aún a día de hoy, debido en parte a la informalidad del entorno, esta terminología es bastante difusa y sus usos no están estrictamente delimitados.

Un *bug* es generalmente algo indeseado tanto para el programador como para el usuario ya que el mal funcionamiento de la aplicación puede degradar la experiencia dependiendo de la gravedad de éste, que puede ir desde un pequeño error gráfico que puede ser inocuo, hasta un error crítico que haga que la aplicación termine de forma inesperada [1], un *crash*, que cause pérdidas en el progreso del usuario. Sin embargo, debido a la gran variedad de situaciones en las que se puede presentar a veces es complicado diferenciar si un comportamiento es intencionado o un *bug*. Existe la posibilidad de que las reglas del juego, estando bien definidas, permitan que el jugador resuelva los retos presentados de diversas formas haciendo uso de su creatividad. Esto es lo que se llama **jugabilidad emergente** y, dado que algunos juegos fomentan este tipo de estrategias, puede o no ser intencional [2].

Por otro lado existe el término ***glitch*** que, aunque a veces se usa de forma indistinta de *bug*, suele usarse para errores menores que no afectan a la jugabilidad gravemente, como puede ser un error gráfico o de sonido que ocurre en un determinado momento, relegando *bug* a aquellos que tienen un impacto más negativo a la experiencia de juego. Sin embargo, puede que un *glitch* tenga un efecto del que el jugador puede beneficiarse, en ese caso se conocen como **exploits** y, a diferencia de los demás, este último es un tipo de jugabilidad emergente no intencional, por lo que es el que más nos interesa y será el foco principal del trabajo. Cabe destacar que el término *exploit* no es muy consistente y a menudo se usa simplemente *glitch* en su lugar, en parte porque hay otros dos casos en los que se utiliza: si una mecánica tiene un fallo de diseño que permite ser abusada, permitiendo al jugador una ventaja desproporcionada; y, por influencia de su uso en seguridad, cuando el *exploit* permite tomar en cierto grado el control del sistema, ya sea local o remoto. Mientras que este último caso sigue siendo un *exploit*, en muchos casos no se puede realizar mediante el juego y puede que para ello requiera ser modificado o usar herramientas externas, ya sea mediante *software* o *hardware*.

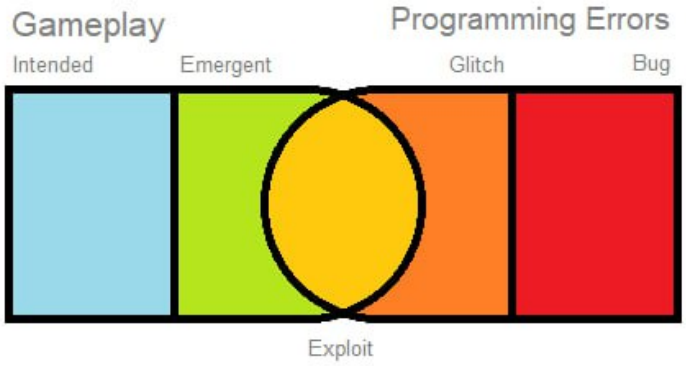

Figura 1: Diagrama que muestra cómo intersectan los términos que vamos a utilizar.

La percepción de un *bug* varía dependiendo de la persona, del efecto que tiene sobre el *gameplay* y del contexto en el que ocurre. En este caso queremos enfocarnos en cómo se percibe un *exploit*, por lo que nos referimos exclusivamente a la parte aprovechable del bug y en el punto en el que el jugador ya cuenta con el conocimiento de cómo usarlo y, probablemente, de cómo replicarlo. El impacto en el *gameplay* es un factor muy importante y puede ir desde algo inocuo, pasando por verlo como una mecánica más, hasta verlo como hacer trampas. Pero debido a que sigue siendo subjetivo, vemos tendencias dependiendo del tipo de juego y cómo los jugadores interactúan con él.

- **Un sólo jugador (Singleplayer)**
  Ya que la decisión de usar el *exploit* recae sobre un único jugador, en un entorno aislado su impacto es positivo o neutral. Al compartir el conocimiento con otros jugadores, puede generarse una discusión sobre ello.

- **Multijugador Cooperativo**
  En los juegos cooperativos suele solucionarse con un acuerdo entre los participantes, por lo que el impacto es similar al de un juego individual.

- **Multijugador Competitivo**
  Cuando un videojuego implementa un sistema de competición entre jugadores a tiempo real los *exploits* suelen ser vistos como algo negativo ya que pueden dar una ventaja injusta si una de las partes no tiene conocimiento sobre ello y esto se ve agravado cuando la jugabilidad es asimétrica (p.e. si el juego tiene múltiples personajes y el *exploit* es sólo posible en uno de ellos. Esto es aún peor si la condición es aleatoria o predefinida). Cuando este tipo de *exploit* se conoce se suelen tomar medidas en contra de su uso, como prohibirlos en eventos competitivos o *bans* en juegos *online*. En la actualidad, es común que los desarrolladores actualicen el juego eliminando el *bug* que lo origina.

- **Tablas clasificatorias (Leaderboards)**
  Hay juegos que siendo de un sólo jugador o cooperativos tienen implementada una tabla de clasificación ya sea local (hay ejemplos tan tempranos como las máquina recreativas) o en línea. Las tablas contienen habitualmente una valoración de la partida calculada a partir de las estadísticas del jugador (p.e. un juego de carreras puede valorar el tiempo que se tarda en recorrer un circuito. En Tetris la puntuación se calcula dependiendo del número de líneas completadas con multiplicadores si se completan líneas simultáneamente.) La percepción en este tipo de competición es similar a la de el multijugador competitivo, pero al jugarse de forma separada no tiene por qué haber forma de comprobar si una partida se ha jugado de forma normal, usando *exploits* o directamente *hacks*, aunque cuando hay diferencias de tiempos demasiado grandes o imposibles se puede intuir. En estos casos en los que no se registra la partida es incluso más frustrante para el jugador y difícil de arreglar para el desarrollador, por lo que desincentiva severamente la competencia.

- **Retos auto impuestos**
  Aunque algunos juegos no están pensados para competir, muchos juegos han desarrollado comunidades que compiten en retos planteados por ellas mismas. El más popular se conoce como ***speedrunning*** y consiste en completar total o parcialmente un juego en el menor tiempo posible. Aunque los primeros casos se dieron en revistas de videojuegos a las que los *speedrunners* enviaban sus tiempos con una foto como prueba, no es hasta la expansión de internet que se populariza este tipo de competición. A día de hoy las comunidades dedicadas al *speedrunning* mantienen sus propias tablas clasificatorias o *leaderboards* y moderan las partidas que presentan los jugadores principalmente en vídeo.

  Las primeras *leaderboards* (Twin Galaxies [3], que era la principal organización dedicada a registrar récords) no permitían el uso de *exploits*. A medida que aparecieron otras *leaderboards* empezaron a usarse de forma extensa, creándose una gran variedad de categorías de competición para distintos juegos. Tanto es así que el nombre de las categorías más comunes suele componerse por el objetivo del *speedrun*, típicamente y si el juego lo permite, 100 % o Any % indicando si se requiere completar todos los objetivos del juego o no; seguido de *Glitchless* si no se permiten los *exploits*, por lo que a día de hoy su uso es esperable.

  El *speedrunning* es muy popular pero no en el mismo nivel que los *esports* debido en parte al alto nivel técnico y de habilidad que requiere para competir. Aún así hay *speedrunners* profesionales gracias a su retransmisión en plataformas como YouTube y Twitch. Cabe destacar que hay

también eventos dedicados al *speedrunning* siendo los más populares los organizados por *Games Done Quick* (GDQ) que a fecha de hoy han recaudado más de $43M para diversas ONG [4].

Pero hay otros retos además del *speedrunning* en las que son importantes, de hecho hay comunidades de *Glitch-Hunters* que se dedican a ello por el simple hecho de llevar el juego al límite. Estas comunidades suelen estar muy relacionadas entre sí y muchos de los descubrimientos de estos son usados por los *speedrunners* para mejorar las estrategias y reducir los tiempos. Algunos se dedican a hacer lo que se conoce como *Tool Assisted Speedrun* [5] (TAS) que consiste en usar herramientas externas para obtener el *speedrun* perfecto teórico, habitualmente imposible para un humano por la precisión requerida, con el simple objetivo de demostrar el límite alcanzable. El TAS más popular a día de hoy, que además es un reto distinto al *speedrun* normal, es el *A Button Challenge* (ABC) de Super Mario 64 de la Nintendo 64, que consiste en completar el juego presionando el botón A el menor número de veces, limitando de esta forma las posibilidades de movimiento del jugador. Tanto es así que el vídeo más conocido sobre este reto, publicado por *Pannenkoek2012* [6], el principal promotor de este reto, tiene más de 5 millones de visualizaciones y existe un documental [7] en YouTube de más de 5 horas explicando la historia detrás del reto hasta inicios de 2023.

Como vemos, en este último caso, hay jugadores que han hecho de los *exploits*, no sólo un pasatiempo por ocio, sino que para algunos es incluso una parte integral de su trabajo. Los desarrolladores, por su parte, a menudo intentan corregir estos *bugs*, especialmente, como hemos visto, en entornos competitivos donde pueden deteriorar la experiencia de juego. En estos casos también tiene su parte positiva, pues si la comunidad es capaz de replicar y documentar el *bug*, es mucho más sencillo encontrar y reparar el problema. Es por eso que se dan casos en los que jugadores que hacen aportes importantes en la comunidad son contratados por su experiencia adquirida, ya sea como *testers* o desarrolladores, convirtiéndose incluso en una forma de cazar talento para la empresa desarrolladora. Estos han tenido influencia incluso en remasterizaciones e incluso adaptaciones de juegos en los que los desarrolladores han dejado *exploits* a propósito para contentar a los jugadores, un caso interesante es el de *Halo 2: Anniversary* [8] que incluso reintrodujeron de forma intencional más *exploits* de la anterior entrega.

En este documento hemos elegido explorar Minecraft, el videojuego más vendido de la historia a día de hoy y con mayor impacto social hasta el punto de ser usado a menudo en entornos educativos [9]. Siendo principalmente un juego de género *sandbox* [10], se basa en fomentar la creatividad del jugador ofreciendo un grado de libertad casi absoluto que le permite modificar a placer un mundo generado proceduralmente formado por bloques. Además introduce mecánicas de supervivencia en el modo de juego por defecto, principalmente salud, hambre y enemigos, con el propósito de dar un objetivo a corto plazo lo hace un entorno excelente para el desarrollo de la jugabilidad emergente y la posibilidad de que aparezca gran variedad de retos. Sin embargo, el juego de por sí no tiene un objetivo definido ya que incluso es opcional encontrar el dragón del *End*, un enemigo que al ser derrotado te lleva a una pantalla de créditos, por lo que se le considera a menudo como el jefe final. El juego nunca te da direcciones explícitas de ello y tras los créditos simplemente puedes seguir jugando, pero se ha convertido en el objetivo perfecto para *speedrunners*. Con la posibilidad de jugar en línea de forma cooperativa también han aparecido modalidades competitivas desarrolladas por la propia comunidad, por lo que todos los casos de interacción entre jugadores que hemos visto anteriormente se dan en este título.

Una peculiaridad es que, debido a la naturaleza del juego, es más común de lo normal que los jugadores exploten las mecánicas para su propósito. Por ejemplo, el primer objetivo que el jugador quiere llevar a cabo es conseguir comida para no morir de hambre, una tarea fácil de llevar a cabo con los métodos más simples. Sin embargo, en el momento en el que el jugador se proponga un objetivo distinto, como puede ser construir un refugio, el hambre se convierte en una molestia con la que tiene que tratar constantemente. Conforme más elaborado sea el objetivo, más se esforzará en buscar una forma de conseguir comida repetidamente con el mínimo esfuerzo para dedicar tiempo a lo que le interesa. Esto ha hecho que los *exploits* sean vistos como algo relativamente normal en Minecraft

incluso para jugadores poco experimentados, especialmente gracias a la simplicidad del juego y los videotutoriales que abstraen a estos de los conceptos necesarios. De esta manera, en lugar de por competición, Minecraft ha desarrollado una comunidad basada en explotar las mecánicas del juego con el propio hecho de explotarlas como único propósito. Para definir esta forma de juego, la comunidad ha adoptado el término Minecraft Técnico, derivado a partir del lema "Minecraft Done Technical" del YouTuber EthosLab [11].

Minecraft es un juego que recibe actualizaciones constantes y, a través de su lanzador, permite jugar en cualquiera de sus versiones, incluso versiones incompletas que se publican para que la comunidad pueda probar los nuevos cambios y adiciones llamadas *snapshots*. Aunque lo más habitual es jugar la versión actual, hay motivos como la nostalgia, querer experimentar las diferencias, compatibilidad con servidores o *mods*, participar en competiciones, completar retos auto-impuestos o abusar de *exploits*, por los que muchos jugadores deciden jugar en versiones anteriores. Los mundos creados en versiones anteriores se pueden actualizar a las nuevas, pero los cambios en las mecánicas del juego a menudo hacen obsoleto el trabajo de los jugadores técnicos. Esto les obliga a renovarse constantemente, por lo que es frecuente que se mantengan en versiones anteriores durante largos periodos de tiempo. Además, la comunidad mantiene investigaciones en una amplia variedad de versiones con el objetivo de documentar [12] y encontrar nuevos *exploits*. Entender uno de los descubrimientos recientes es la principal motivación para haber escogido Minecraft como tema para este proyecto, ya que creemos que es relevante para la historia de los *exploits* en los videojuegos dadas sus características únicas.

Este documento está estructurado de la siguiente forma tras la introducción:

- **Fundamentos y Estado del Arte**
  Vamos a echar un vistazo a algunos de los *exploits* más relevantes que han surgido en la historia de los videojuegos y la historia detrás del que queremos estudiar.
- **Metodología y Objetivos**
  En esta sección veremos qué queremos lograr en este documento y cómo vamos a conseguirlo, además de las herramientas que vamos a utilizar para ello.
- **Evaluación y Discusión**
  Dada la densidad del tema que vamos a tratar hemos decidido separar esta sección en tres partes diferenciadas. La primera parte consiste en explorar las mecánicas base del juego y algunos *exploits* requeridos que no son necesariamente exclusivos del tema principal. En la segunda parte exploraremos el *exploit* en cuestión, la estrategia para lograrlo y sus aplicaciones. Por último exponemos nuestro análisis y propuestas para la mejora de la estabilidad y eficiencia del *exploit*.
- **Conclusiones y Trabajo Futuro**
  Finalmente, ahora que hemos entendido cómo funciona y dados los resultados que hemos obtenido veremos qué más se puede hacer para mejorarlo.

2 Fundamentos y Estado del Arte

Los *bugs* y *exploits* en videojuegos no son algo reciente, llevan existiendo desde la concepción de los videojuegos. Para comprender la relevancia de los mismos, es necesario entender cómo han ido evolucionando.

En cada época, debido a los cambios de los procesos de desarrollo y plataformas usadas, podemos observar distintos tipos de bugs. En las épocas más sencillas, los bugs y exploits son sutiles, pero, a veces, fatales. Por contra, en las épocas más modernas, debido a la proliferación del uso de motores de videojuegos más grandes y más depurados, estos fallos quedan relegados a problemas menores.

Hemos dividido las épocas (o eras) de los videojuegos en 4 grupos, dónde distinguiremos el inicio de los videojuegos (Retro Era), la época 2D de los videojuegos (Pre 3D era), el inicio y desarrollo de los mundos tridimensionales (3D Era), y el estado actual de la industria, con juegos de todo tipo.

2.1 Retro Era

Los primeros juegos eran muy simples mecánicamente, por lo que era difícil encontrar errores. Los más populares eran máquinas recreativas (arcade) ya que las consolas y ordenadores eran caros y poco potentes. Tampoco hay mucha información pues, sin internet, la existencia de estos *bugs* no se difundían tan fácilmente. Algunos ejemplos destacables son:

2.1.1 Space Invaders (1978)

Un juego de arcade clásico que consta de una nave que se puede mover sólo en el eje horizontal mientras un grupo de enemigos al que hay que disparar desciende lentamente. A medida que el jugador derrota enemigos, el juego se acelera debido a la reducción de carga que supone para el procesador. Su creador, Tomohiro Nishikado, se dio cuenta del *bug* [13] pero decidió dejarlo como un reto para el jugador, siendo uno de los primeros casos de dificultad adaptativa.

2.1.2 Asteroids (1979)

Otro juego de arcade, en el que controlas una nave que cuando sale de pantalla aparece en el borde contrario. Como dice el nombre, el objetivo es esquivar y destruir asteroides que se mueven por la pantalla, y platillos que disparan al jugador, aunque no pueden hacerlo a través de los bordes de la pantalla. Los asteroides aparecen por oleadas, por lo que hasta que no destruyes todos los de una oleada no vuelven a aparecer, mientras que los platillos siguen apareciendo ocasionalmente. Podemos abusar de estas reglas dejando un solo asteroide y jugando en las esquinas de la pantalla, lo que nos permite esquivar fácilmente el asteroide restante a la vez que aumentamos la cantidad de puntos de forma indefinida. Esto se conoce como "*lurking exploit*" y es un *exploit* de diseño pues no hay un error de programación *per se*, el cual acabó siendo parcheado en una "actualización" del juego [14].

2.2 Pre 3D Era

Una vez explotan las consolas, la industria empieza a desarrollar juegos más elaborados en lugar de los casuales y cortos de las arcade. Esto hace que los jugadores dedicaran más tiempo al mismo juego lo que, junto a la aparición de mecánicas de juego más complejas, van descubriendo un gran abanico de *glitches* con efectos más o menos graves.

2.2.1 Super Mario Bros. (1985)

Un juego de plataformas lanzado para la Famicom y la NES (*Nintendo Entertainment System*), dos versiones de la misma consola de Nintendo, en el que controlamos al popular personaje Mario dónde el objetivo es moverse a través de varios niveles esquivando obstáculos y derrotando enemigos para salvar a la princesa Peach que ha sido raptada por el malvado Bowser. Este juego es conceptualmente

mucho más complicado que los anteriores, el movimiento, los enemigos y el mundo tienen un mayor nivel de detalle y diferentes formas de interactuar.

Uno de los *glitches* que ocurren es precisamente en la interacción de Mario con el mundo. Cuando Mario se mueve, este lo hace varios píxeles en cada fotograma o *frame*. Si se mueve dentro de un muro, el juego intenta empujarlo fuera de éste moviéndolo en la dirección contraria a la que esté pulsada en el mando. Cambiando de dirección en el mismo *frame* (*frame perfect*) hace que el juego empuje a Mario dentro de este. Los *glitches* que involucran cómo el juego maneja las colisiones permitiendo al jugador atravesar paredes son un caso muy común conocido como *clipping*. Si al *clippear* una pared entramos en lugares que normalmente no son accesibles se dice que estamos *Out of Bounds* (OoB), del inglés fuera de los límites. Estos *glitches* pueden ocurrir de muchas formas distintas, en este caso ocurre debido a un desperfecto en la lógica.

El juego presenta a menudo unas tuberías que funcionan como *warps*, transiciones entre niveles o zonas del mismo. La mayoría de *warps* presentes en el juego (hay algunos que no son tuberías) determinan el lugar de destino usando una única variable en memoria que varía según la posición horizontal de la pantalla en lugar de la posición de Mario. Hay varias formas de desincronizar estas posiciones, siendo una de ellas ser empujado dentro de una pared, lo que puede hacer que un *warp* lleve a Mario a un lugar incorrecto. Este es otro tipo de *glitch* habitual conocido como *Wrong Warp* que a menudo está causado por una inicialización incorrecta de la variable que lo controla [15].

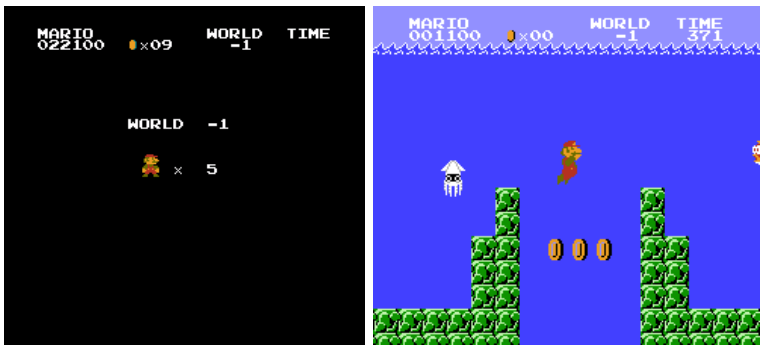

Figura 2: “Nivel -1” en Super Mario Bros.

Uno de los *glitches* más famosos de la historia de los videojuegos es el *Minus World* [16] o mundo negativo. En realidad son mundos con un identificador inválido y reciben este nombre porque al mostrarse en pantalla aparecen como “nivel -1” (Figura 2). Esto es debido a que el primer carácter se corresponde con el identificador y la textura es a menudo invisible. Debido a cómo está implementado, estos mundos están formados por combinaciones de otros mundos y datos no relacionados que se interpretan como parte del nivel [17]. Este tipo de *bug* se conoce como **corrupción de memoria**. Una manera de llegar hasta él, es precisamente a través de un *Wrong Warp*, aunque de forma ligeramente distinta a la que hemos explicado.

2.2.2 Pokémon Red/Green/Blue/Yellow (1996-1998)

La primera generación de los juegos de Pokémon para la Game Boy. Es un JRPG (*Japanese Rol-Playing Game*) por turnos que tuvo un gran impacto y dio a la marca una relevancia que mantiene a día de hoy, siendo la franquicia multimedia más rentable del mundo [18], por encima de las dos franquicias más exitosas de Disney (*Mickey Mouse & Friends* y *Star Wars*). El juego presenta unas criaturas llamadas Pokémon a las cuales debemos entrenar para que combatan entre ellas con el objetivo de competir en una liga con los mejores entrenadores de la región. Como otros juegos RPG, presenta un mundo con variedad de personajes (*Non-Playable Characters* o NPC) con diálogos, encuentros aleatorios, un equipo formado por múltiples Pokémon y un inventario de objetos con diferentes propósitos. Los primeros juegos de Pokémon son conocidos por estar muy mal depurados, con grandes cantidades de *glitches* que han llevado a crear comunidades que documentan y exploran los límites del juego.

Otro de los *glitches* más famosos de los videojuegos es MissingNo. (abreviatura de *Missing Number*) que es el nombre que el juego da algunos de los diferentes "Pokémon Glitch", aquellos con un identificador inválido [19] (Figura 3). De la misma forma que los *Minus World*, estos leen todos sus datos de posiciones de memoria no intencionadas, por lo que su aspecto, sonido y movimientos parecen rotos. Hay diversos *glitches* que nos permiten obtener a MissingNo., siendo el más conocido el uso de un espacio de memoria sin inicializar a causa de un error en el mapeado, pero el hecho de encontrarlo tiene efectos interesantes.

Figura 3: Un encuentro típico con MissingNo.

El juego incluye un bestiario llamado Pokédex que incluye una entrada para cada Pokémon que almacena si se ha encontrado y capturado con anterioridad. Sin embargo, el índice en la Pokédex no es igual al que el Pokémon usa internamente, por lo que cada uno almacena qué entrada le corresponde. Tras un encuentro con MissingNo. la Pokédex debe actualizarse, tomando su índice también de memoria no relacionada. Para todas las variantes este coincide con la entrada 0 que el código encargado de actualizar la Pokédex interpreta como un *offset* de 256, lejos de la sección de memoria utilizada por esta. Esto es otro tipo de corrupción de memoria, específicamente un desbordamiento de búfer o *buffer overflow*.

A su vez, esta posición resulta cuadrar con el último bit perteneciente a la cantidad del objeto almacenado en el hueco número 6 del inventario del jugador, incrementando su cantidad en 128 si era 127 o inferior [19]. Por tanto, también es un método de duplicación (*duping*), otro tipo de *glitch* muy común que está presente en juegos que incluyen un inventario; y, al sobrepasar la cantidad máxima en que se puede agrupar un objeto[1], en este caso 99, hemos causado lo que se conoce como *overstacking*, un efecto secundario común de la duplicación.

Tener objetos en el inventario cuya cantidad es 255 también es útil pues el juego utiliza este valor como marca de fin, a la vez usa un contador para, entre otras cosas, determinar cuántos objetos deben mostrarse. Manipular estos objetos se puede utilizar para provocar un *(arithmetic) underflow* en el contador, permitiéndonos ver mucho más allá de nuestro inventario pues normalmente está limitado a sólo 20 objetos. Este *glitch* nos permite manipular la memoria expuesta a placer usando las acciones habituales del inventario incluyendo obtener objetos inválidos [20, 21].

Muchos objetos en Pokémon se pueden usar desde el inventario y qué hace cada uno es determinado por una tabla de funciones. Como es de esperar, los objetos inválidos acaban leyendo memoria no relacionad fuera de la tabla, pero, a diferencia de las ocurrencias anteriores, esta se interpreta como punteros a memoria que se va a ejecutar como si fuera código. En muchos casos la llamada acabará *crasheando* el juego por diversos motivos, pero en otros, puede que intente ejecutar en zonas de memoria que podemos manipular. Esto es lo que se conoce como **ejecución de código arbitrario** o **ACE** (*Arbitrary Code Execution*), el tipo de *exploit* más potente pues permite al jugador tomar control, habitualmente absoluto, sobre la ejecución [22].

[1] Este concepto es conocido como *item stack*, del inglés "pila de objetos", y el límite puede ser variable entre distintos objetos.

2.3 3D Era

Con la salida de consolas más potentes como la PlayStation o la Nintendo 64 empezaron a desarrollarse juegos con técnicas de dibujado 3D y empezaron a usarse lenguajes de más alto nivel en lugar de ensamblador. Esta época se caracteriza en que ya no aparecen *glitches* de corrupción de memoria tan a menudo debido a una mayor presencia de protección de memoria, pero debido al incremento en complejidad, los errores de lógica y diseño persisten.

2.3.1 Super Mario 64 (1996)

Otro de los juegos más populares de la franquicia y su primer juego 3D. Es un juego de plataformas de mundo abierto que se caracteriza por el diverso conjunto de movimientos que Mario puede realizar. Esta entrega tiene el mismo objetivo y trama que el original pero, al estar pensado como un juego de mundo abierto, varias puertas bloquean el avance del jugador por el mundo. Para abrirlas, Mario necesita recoger cierta cantidad de estrellas que están repartidas por el juego o llaves que se obtienen al derrotar a Bowser en un par de encuentros obligatorios antes de la pelea final.

La diversidad y complejidad de los movimientos de Mario hizo que este juego se popularizara entre los *speedrunners* tanto que aún a día de hoy es uno de los más populares [23]. La mayor parte de una partida consiste en recoger las estrellas y llaves necesarias, por lo que, si se pudieran ignorar los requisitos, supondría una reducción drástica en la duración del *speedrun*. Esto es lo que se conoce como *sequence breaking*, una categoría de *exploit* muy común en variedad de juegos en la que se agrupan aquellos que modifican el orden o eliminan parte de los eventos principales del juego, siendo especialmente visible cuando la narrativa se vuelve inconsistente o se tienen elementos antes de tiempo. La forma de lograrlo consiste en atravesar la puerta abusando de cómo se manejan las colisiones de una entidad. Cuando una entidad se mueve, el juego divide el movimiento en múltiples pasos de forma que sólo interactúa con los elementos con los que colisione en cada uno de ellos. Si Mario alcanza una velocidad lo suficientemente grande como para que un obstáculo se encuentre entre dos de estos pasos, este es ignorado, efectivamente atravesándolo.

La método más conocido para conseguir esta velocidad es con el movimiento conocido como ***Backwards Long Jump*** (BLJ) [24, 25]. Este consiste en aprovechar que el juego aplica un multiplicador a la velocidad horizontal de Mario cuando este hace un salto largo. Para ello necesita tener suficiente velocidad horizontal inicial, pero al tocar el suelo tras el salto, se inicia una animación durante la cual se puede repetir el movimiento sin esta restricción. Además se aplica un tope tras el multiplicador para evitar que esta crezca indefinidamente al repetir el movimiento. Debido a que se permite el control aéreo al jugador durante el salto, cuando la velocidad inicial es lo suficientemente baja se puede invertir la dirección del movimiento manteniendo el *joystick* en la dirección opuesta, resultando en un BLJ. El problema surge finalmente de que el tope de velocidad sólo se aplica cuando esta es positiva, por lo que nos permite multiplicar la velocidad acumulada una y otra vez.

Mario es capaz de llevar consigo algunos elementos del juego, como puede ser una caja. Cuando Mario agarra un objeto, en realidad no lleva el objeto de verdad consigo, sino una referencia al hueco que ocupa en la tabla de objetos cargados. Mientras que el objeto original se vuelve invisible, intangible y no puede ser descargado, lo que se conoce como un estado de limbo, en las manos de Mario se dibuja una copia visual de este. Al realizar la acción de lanzar o dejar el objeto, el original sale del limbo y se coloca en la última posición de la copia visual o HOLP (*Held Object's Last Position*). Sin embargo, hay una ventana de dos *frames* entre que se ejecuta la acción de agarrar[2] y el objeto entra en el estado de limbo. Si el objeto se descarga durante este tiempo, el hueco de la tabla queda libre para que otro objeto ocupe su posición. Este tipo de *glitch* es lo que se conoce como *stale pointer*, que viene a significar "puntero obsoleto", que a su vez es un tipo de referencia colgante o *dangling pointer*.

Esto permite a Mario llevar consigo objetos que normalmente no puede y, al no haber pasado por el proceso normal, estos no están en el estado de limbo, por lo que pueden volver a descargarse y ser reemplazados por otros. Aunque visualmente parece que son dos objetos distintos, cuando Mario

[2] Hay tres acciones distintas que permiten agarrar un objeto, pero sólo existe esta desincronización al dar un puñetazo.

Figura 4: Clonando goombas en Super Mario 64.

suelta el objeto, el juego desplaza el original de la misma manera que cualquier otro. Sin embargo, cuando ocurre esto se intenta llamar al *script* que define su comportamiento tras esta acción. Al carecer de uno, el comportamiento del objeto es sobrescrito, por lo que queda congelado en el HOLP y nunca se descarga ya que no puede actualizarse. En este estado, cuando Mario colisiona con un objeto aún llama al *script* encargado de la interacción. Sin embargo, muchas de estas interacciones ocurren parcialmente porque requieren que el objeto se actualice para completar la acción, por lo que, a menudo, sólo se pueden ejecutar una vez. Por ejemplo, si el objeto que hemos lanzado es una moneda, al tocarla aumenta el contador de monedas recogidas, pero no desaparece y ya no se puede interactuar con ella. Si el objeto pertenece además a un *spawner*, un objeto invisible que se encarga de manejar la carga y descarga de otro grupo de objetos, este no recibe información de la interacción y, por tanto, este creará un nuevo objeto al ser recargado. Este es otro *exploit* conocido llamado *cloning* [26], que representa el mismo concepto que *duping* pero afecta a elementos del mundo en lugar de a objetos del inventario (Figura 4).

2.4 Actualidad

En la actualidad han proliferado los motores de videojuegos, entornos de desarrollo dedicados a crear videojuegos que incluye herramientas y librerías que permiten simplificar el desarrollo ofreciendo distintas funcionalidades como el sistema de dibujado, físicas, colisión, sonido, animación o red, entre otros. Aún así, aquellos que usan un motor propio, están programados generalmente en lenguajes de alto nivel como C#, Java o Python que abstraen al programador del manejo de memoria y tienen librerías específicas para distintos aspectos. Esto hace aún más difícil que ocurran los errores graves relacionados con memoria y punteros que hemos visto anteriormente e incluso los problemas con mecánicas base del juego.

Figura 5: Out of Bounds en Portal (izquierda) y Portal 2 (derecha).

Debido a todo esto, la mayoría de los *exploits* en los juegos modernos ya los hemos visto en generaciones anteriores, pero vamos a revisar los que persisten. *Clipping* es uno de los más comunes generalmente causado por un error en el cálculo de las colisiones, un caso extremo de mecánicas del juego o que la geometría tenga una vía de escape porque faltan colisiones o una ruta imprevista. Un ejemplo de ello son *Portal* (2007) [27] y *Portal 2* (2011) [28], su secuela, donde el jugador puede atravesar paredes a causa de un error en el posicionamiento del personaje al atravesar portales, la mecánica principal del juego (Figura 5). Otro ejemplo de un juego recientemente actualizado es *World of Warcraft* (2004-2023), dónde múltiples fallos de geometría permiten escaparse del espacio delimitado para el jugador (Figura 6). Como hemos visto anteriormente, esto permite al jugador en muchos casos acceder al espacio fuera del mapa (*Out of Bounds*) e incluso cuando no, habitualmente, permite saltarse parte del juego (*sequence breaking*).

Figura 6: Out of Bounds en la zona principal de *World of Warcraft: Dragonflight* (Noviembre de 2022).

Por otro lado, *duping* y *cloning* sigue siendo una ocurrencia constante en los juegos más recientes, a menudo causados por errores en los menús, como en el caso de *Pokémon Escarlata/Púrpura* (2022) [29] o *The Legend of Zelda: Tears of the Kingdom* (2023) [30], la última entrega a día de hoy de sus respectivas franquicias, que pueden producir este efecto al pulsar dos botones de forma simultánea con unas condiciones específicas.

2.4.1 Minecraft (2011-Actualidad)

El caso de Minecraft es un tanto especial entre los juegos modernos. A lo largo del tiempo se han encontrado multitud de formas de ejecutar los *exploits* que acabamos de mencionar, sin embargo, el tipo de *exploit* más común suele estar relacionado con el diseño de las mecánicas. En los últimos años, una de las versiones más activas en la parte de más alto nivel de la comunidad técnica es *Minecraft 1.12* (2017) y es la que nos vamos a centrar en este documento.

A mediados de 2020 esta comunidad se centra en perseguir la obtención de *bedrock* [31], un bloque indestructible que no se puede conseguir en su forma de objeto ya que sirve por lo general para impedir al jugador caer al vacío debajo del mundo. Exploraron múltiples teorías y, aunque consiguieron su objetivo en poco tiempo [32], una de estas era mucho más interesante que las demás pues se extendía a otros bloques que, por diversos motivos, también eran imposibles de obtener. La teoría consiste

en que ciertos bloques pueden convertirse en una entidad y, si se sustituye por otro bloque en un momento específico, la entidad se crea con ese otro bloque incluso si no puede caer. El problema es que, en la práctica, no había forma alguna de realizar ese cambio. A pesar de todo, unos meses después, un usuario conocido como *coolmann24* consigue ejecutar por primera vez una demostración de cómo podría ser posible [33] usando un nuevo tipo de *exploit* en los videojuegos.

Aunque el juego a penas ejecuta código en paralelo excepto para el dibujado y operaciones de entrada salida, resulta que una mecánica muy específica ejecuta un hilo que simplemente consulta algunos bloques y pasa información al hilo principal para que luego actualice ciertos efectos visuales. Todo este proceso, a pesar de no estar implementado correctamente, no supone un problema de forma normal, pero hay un camino abusable en la ejecución que resultó ser apenas posible. Este es el primer caso que conocemos de explotación de una *data race* en los videojuegos, un tipo de condición de carrera que ocurre cuando al menos un hilo lee y otro escribe la misma localización de memoria, causando que el resultado de acceder a esta sea no determinista, lo cual lleva a comportamientos inesperados, perdidas de rendimiento, o incluso en algunos casos, en lenguajes compilados, que este no se pueda compilar correctamente [34, 35].

Sin embargo, esta demostración era sólo eso, pues estaba fuertemente limitada por sus requisitos, haciéndola inviable en la práctica. Esto fomentó una investigación a fondo durante los siguientes años. No fue hasta finales de 2021, principios de 2022, que la comunidad da con una estrategia que permite ejecutar el *exploit* de forma práctica [36]. Todo el área de conocimiento relacionada con este *exploit* fue bautizada como *Threadstone*[3] y se desarrollaron multitud de mejoras y nuevos usos. De entre estos, además de ser el más común a día de hoy, destaca por usar más de un hilo además del principal para conseguir dos condiciones de carrera simultáneas.

[3] Es un juego de palabras entre *thread* (hilo) y *redstone*, un material del juego que funciona como conductor de corriente que comparte el nombre con este. Este término es también usado por la comunidad para referirse al conocimiento sobre mecanismos. Además es una broma que referencia a la nomenclatura dada por la comunidad a la hora de categorizar algunos mecanismos, como la *Slimestone* o *Leafstone*.

3 Metodología y Objetivos

En este trabajo tenemos la intención de explicar por qué el caso de Minecraft es especial en comparación con los *exploits* que se han descubierto anteriormente en otros videojuegos y por qué es posible que no vuelva a darse nunca más un *glitch* de características similares. Para ello, primero hemos dado un vistazo a algunos de los videojuegos más prolíficos en este aspecto, viendo algunos de los *glitches* más icónicos, habitualmente presentes en varios títulos, con el objetivo de ver cómo han ido variando, tanto en complejidad como utilidad, dependiendo de las limitaciones de los sistemas y los avances tecnológicos.

La edición original de Minecraft, ahora conocida como *Minecraft: Java Edition*, es una aplicación que se ejecuta en un sistema moderno y está programada en Java, un lenguaje que abstrae al programador de la gestión de memoria. Esto hace que carezca de los problemas causados por la ausencia de protección de memoria y de *exploits* relacionados con punteros obsoletos presentes en los títulos de consolas clásicas, por tanto es mucho más complicado que podamos llegar a ejecutar código arbitrario. Por el lado contrario, no usa un motor de videojuegos como la mayoría de los juegos actuales, sino que implementa el suyo propio (en parte debido a que en su momento no había ninguno que se ajustase a las características del juego), por lo que es más fácil que hayan vulnerabilidades graves. El motivo por el cual hemos escogido hablar de Minecraft como caso de estudio es que la comunidad ha descubierto recientemente una serie de *exploits* única hasta la fecha que, a pesar de la robustez adquirida por la mejora tecnológica, en complejidad y ejecución se asemeja más a los presentes en títulos antiguos, aún siendo fundamentalmente distinta.

Para poder entender el caso de Minecraft primero necesitaremos estudiar cómo funcionan ciertos aspectos del juego en profundidad como pueden ser algunos bloques relevantes, cómo se procesa el mundo y cómo podemos interactuar con él; así como algunos otros *exploits* necesarios que ya eran conocidos anteriormente, como es el *savestate* o la cancelación de actualizaciones. Una vez hayamos establecido las bases, exploraremos cómo ocurre el *exploit* que queremos tratar y algunos de sus usos más importantes enfocados en obtener ciertos elementos del juego que serían imposibles de otra forma. Por último veremos que el juego es extremadamente inestable mientras trabajamos con este *exploit*, por lo que podemos tener que repetir parte del procedimiento o perder todo el progreso si el juego acaba *crasheando*. Nuestro propio aporte consistirá en investigar el porqué de estos problemas y a ser posible encontrar soluciones, mejorando la estabilidad y eficiencia de la ejecución del *exploit*.

Para ello analizaremos las trazas de los *crashes* y usaremos la versión decompilada del juego que nos interesa. Existen múltiples herramientas que permiten decompilar el *bytecode* de Java, pero en este caso está también ofuscado con una herramienta llamada *ProGuard* [37]. Aunque las versiones modernas de Minecraft proporcionan un mapa de ofuscación, no es el caso para la versión en la que estamos trabajando. Afortunadamente la comunidad ha desarrollado herramientas para ello aunque el mapeado de nombres haya sido realizado parcial y manualmente. Con todo esto podremos leer y modificar el código para analizar a fondo qué ocurre en el juego.

A continuación detallamos qué *hardware* y *software* hemos utilizado:

- **Versión de Minecraft:** Minecraft 1.12.0
  Lanzada en 2017, una de las versiones más populares en la comunidad técnica y en la que ocurre el *exploit* que vamos a investigar.

- **Decompilador/Deofuscador:** MCP 940 para Minecraft 1.12.0
  Mod Coder Pack (MCP) [38] es una herramienta que permite decompilar, deofuscar y reofuscar Minecraft: Java Edition. Desarrollada por, entre otros, Michael "Searge" Stoyke, actual Tech Lead de Minecraft: Java Edition; y Thomas "ProfMobius" Guimbretière, ahora ex-desarrollador de Minecraft: Java Edition; antes de que tuvieran estos cargos.

- **Versión de Java:** Java 1.8.0_261 64bit
  Vamos a analizar código de algunas clases implementadas en la JDK extraído de esta versión,

aunque estas no varían demasiado entre versiones. Se han realizado pruebas en Java 1.17.0 sin problemas.

- **Librerías:** Minecraft lleva integradas las librerías necesarias que vamos a analizar. Aunque no deberían variar, hemos trabajado con las que utiliza a fecha de Septiembre de 2022. En concreto vamos a ver parte de la implementación de FastUtils 7.1.0.
- **Sistema Operativo:** se han hecho pruebas en Windows 10 (22H2) y Arch Linux (Kernel 6.3.3).
- **Procesador:** AMD Ryzen 7 3700X 8-Core Processor.

Además, para posterior análisis hemos desensamblado parte del código ejecutando el juego con los siguientes argumentos de la JVM:

```
-XX:+UnlockDiagnosticVMOptions -XX:CompileCommand=print,net.minecraft.util.BitArray::setAt
```

4 Funcionamiento del Juego

En primer lugar vamos a analizar paso por paso, el funcionamiento interno del juego. Esto resultará relevante más adelante.

4.1 Tick

En Minecraft, el bucle principal de juego itera 20 veces por segundo ($50ms$) y cada iteración se conoce como un tick. Debido a que el juego está separado en cliente y servidor, mientras que el rendimiento del cliente se mide de la forma habitual, en ciclos de dibujado, *frames per second* ($60fps$); para el servidor se utilizan dos medidas: *ticks* por segundo ($20tps$), que sólo puede bajar si se estresa mucho el tick; y milisegundos por tick ($50mspt$), que nos permite estimar qué margen hay antes exceder el tick. Durante el tick se procesan cada uno de los componentes del mundo en un orden determinado. Cada una de las divisiones del proceso se conocen en la comunidad como fase.

4.1.1 Terreno dinámico

Si bien la actualización de entidades es algo común en los videojuegos, Minecraft permite modificar completamente el terreno y hay una gran variedad de formas en las que puede ocurrir.

4.1.1.1 Synchronized task phase o Player phase

La forma más simple de ver una actualización es un jugador poniendo o quitando un bloque del mundo. Sin embargo, estas acciones pueden afectar a más bloques que el que se ha modificado. Cuando un jugador modifica un bloque el juego propaga esta actualización para que otros bloques adyacentes reaccionen **de forma inmediata** si es posible. Estas reacciones dependen de cada tipo de bloque. Por ejemplo, una antorcha que está puesta encima de un bloque debe destruirse si el bloque de soporte es eliminado. Este tipo de actualizaciones, a su vez deben notificar a los bloques adyacentes de forma **recursiva**.

4.1.1.2 Scheduled update phase

Hay algunos bloques que en lugar de realizar su actualización inmediatamente la programan con un retardo determinado. Esto es lo que conoce como ***scheduling*** o *tile tick*. Por ejemplo, hay bloques que pueden caer generando una entidad temporal (*falling block*), como el bloque de arena. Este tipo de bloque programa la actualización con un tick de retardo, por lo que se actualizará en otro tick sin la interacción del jugador. Cuando finalmente ocurre la actualización, esta va a notificar también a sus bloques adyacentes, por lo que puede provocar cadenas ya sean inmediatas o prolongadas en el tiempo.

4.1.1.3 Block event phase

Existen algunos bloques para los que, dada la complejidad de su funcionalidad, la implementación de bloque no es suficiente. Por ejemplo, un cofre tiene su propio inventario para almacenar objetos pero un bloque no tiene forma de guardar esa información. Para ello el juego crea un tipo de entidad llamada ***tile entity*** que está vinculada a una posición del mundo. Algunas *tile entities* utilizan un sistema de eventos para enviar a los clientes actualizaciones. Cuando se crea un evento se coloca en una lista para mandarlos al final del tick a los jugadores cercanos. Sin embargo, el pistón usa este sistema de forma cuestionable, posponiendo su actualización a este punto. Debido a esto, si un pistón actualiza otro en cadena, en lugar de actualizarse recursivamente se añadirá a la lista.

4.1.1.4 Entity and Tile Entity phase

En realidad el jugador no necesita interactuar con el mundo para que se vea modificado. Hay entidades que pueden destruir y colocar bloques, pero simplemente por el hecho de navegar por el mundo interactuar con bloques que reaccionan a ellos, como pueden ser las placas de presión. Algunas *tile entities* también requieren ser actualizadas a menudo, por ejemplo la tolva (normalmente conocido como *hopper*, es un bloque que puede transferir objetos entre distintas *tile entities* que almacenan objetos), y pueden llegar a modificar el mundo.

4.1.1.5 Random tick phase

Aún más, el entorno puede verse modificado aleatoriamente por eventos meteorológicos, como la formación de hielo o acumulación de nieve; o el simple paso del tiempo, como el crecimiento de un cultivo.

4.1.2 Automatización: Redstone

Minecraft cuenta con múltiples bloques enfocados a interactuar con entidades u otros bloques. Vamos a hacer un uso extensivo de estos bloques para realizar ciertas tareas, por lo que es relevante saber de su existencia.

4.1.2.1 Cableado

Para ello utiliza un mineral llamado *redstone* que, pulverizado, puede colocarse en el mundo actuando como un cable capaz de propagar una energía homónima (*redstone power*). Esta energía es transmitida por diversas fuentes, las más comunes emiten el nivel de energía máximo (15) que va reduciéndose a medida que se aleja de esta, por lo que su propagación por el cable es limitada (Figura 7). Además, esta energía puede pasar a través de un bloque siempre y cuándo sea un bloque sólido (por lo general son bloques completos y que no dejan pasar luz a través, aunque hay excepciones). A través de un bloque el nivel de energía no se ve reducido y puede activar cualquier componente excepto otro cable.

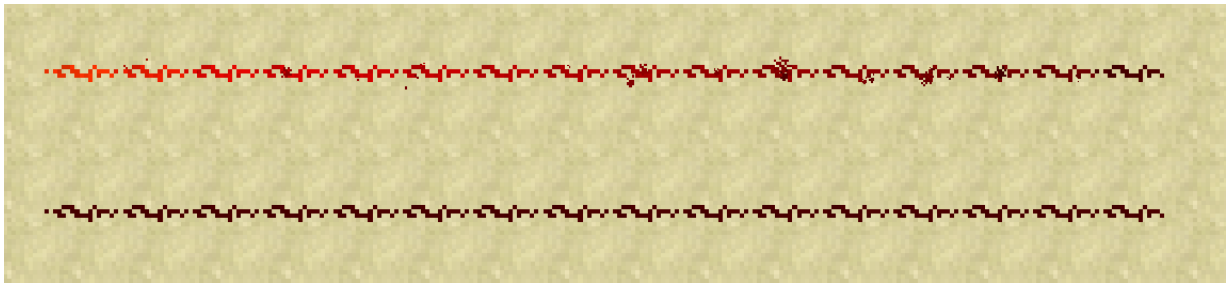

Figura 7: Dos cables de redstone. En el de arriba se puede ver cómo la corriente baja hasta apagarse.

4.1.2.2 Fuentes de energía

Hay fuentes interactivas como el botón, que emite un pulso cuando el jugador interactúa con él; la palanca, que permite cambiar entre apagado y encendido; la placa de presión, que emite señal si hay una entidad encima, etc. Otro bloque muy importante es el *observer*, un bloque direccional que, como su nombre indica, observa los cambios que se producen en el bloque adyacente al que mira. Si detecta un cambio, da un breve pulso de corriente.

El bloque de *redstone* es una fuente estática que da energía constante a los bloques adyacentes. Por otro lado la antorcha de *redstone* se comporta de forma similar pero actúa además como un inversor cuando recibe corriente el bloque sobre el que está colocada, por lo que tenemos suficiente como para poder crear sistemas lógicos completos. Ya que la inversión no tiene en cuenta el nivel de energía que

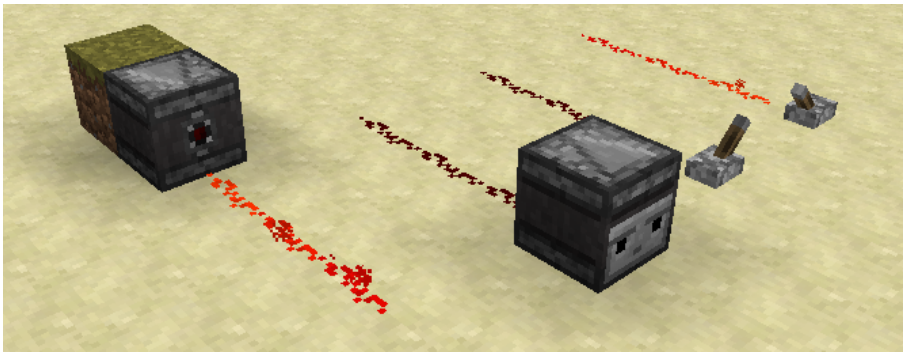

Figura 8: A la derecha dos palancas, una activada y la otra desactivada. A la izquierda, dos *observers*, el de la izquierda acaba de detectar un cambio.

recibe, es decir, emite el nivel 0 si recibe energía y 15 en caso contrario; podemos encadenar inversiones para alargar la corriente indefinidamente.

4.1.2.3 Diodos

Otro elemento importante en el juego son los diodos, que son aquellos bloques que tienen una entrada y una salida de corriente. Actualmente hay dos tipos, siendo el primero el repetidor, que simplifica la extensión de los cables emitiendo el máximo nivel de energía si recibe cualquier nivel distinto de 0 y, a su vez, permite aplicar a la transmisión distintos retardos de forma más sencilla.

Por otro lado está el comparador que es capaz de realizar varias operaciones bastante más complejas. En primer lugar, es capaz de inspeccionar distintos bloques y *tile entities* a través de su entrada principal, emitiendo distintos niveles de energía dependiendo del estado del bloque. Por ejemplo, si inspecciona un baúl, emite un nivel de 0 a 15 dependiendo de cuán lleno esté su inventario. Además cuenta con una segunda entrada que modifica el nivel de la salida dependiendo de en qué modo esté el comparador. Si está en modo comparación el nivel de salida es igual al nivel de la entrada principal siempre y cuando sea mayor o igual que la entrada secundaria. Si está en modo substracción el nivel de salida es igual a la diferencia entre el nivel de la entrada principal y la secundaria con un mínimo de 0.

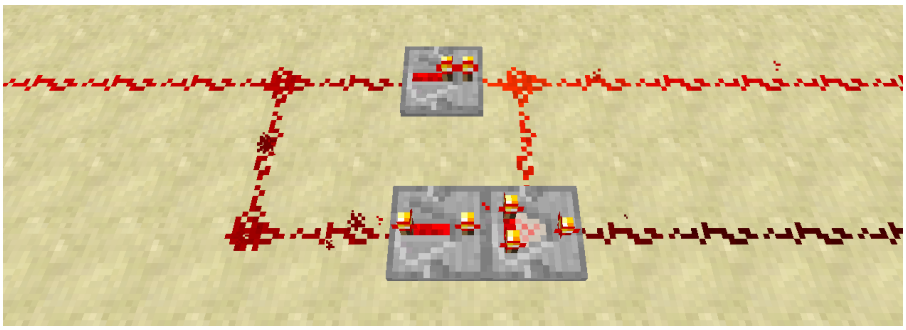

Figura 9: Repetidores y un comparador en modo substracción.

4.1.2.4 Interacción con bloques

El pistón es un bloque que, al recibir corriente, es capaz de extender su brazo y empujar los bloques que tiene delante. Puede mover hasta un máximo de 12 bloques aunque hay excepciones, algunos bloques son destruidos al intentar moverse, mientras que otros son inamovibles. Si encuentra uno de los últimos o se supera el límite, el pistón no se acciona. Por otro lado existe una versión "pegajosa" que es también capaz de tirar del último bloque durante la retracción.

Además, hay un bloque especial llamado *slime block* que es capaz de arrastrar con él otros bloques adyacentes, permitiendo empujar y tirar de grupos de bloques más complejos, manteniendo siempre los límites del pistón.

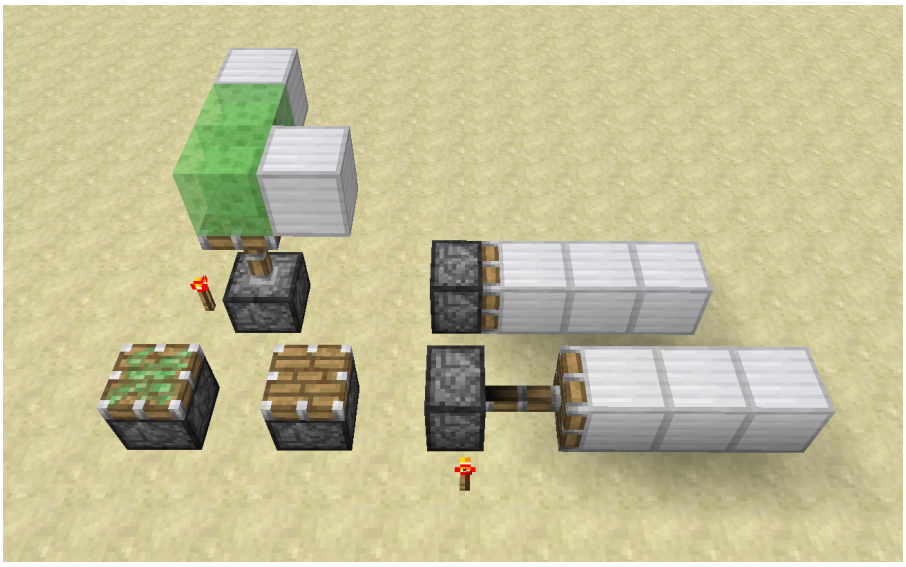

Figura 10: Pistones de ambos tipos y *slime blocks*.

Hay otras maneras de poner y quitar bloques, pero son mucho más específicas.

4.1.2.5 Interacción con inventarios

Hay distintos bloques que pueden interactuar con inventarios, ya sean entidades u otras *tile entities*, siendo los más comunes:

- El *hopper* puede recoger objetos en forma entidad o extraerlos de un contenedor que se encuentre encima de él. Además, puede insertarlos en otro contenedor dependiendo de la dirección en la que apunte.
- El *dropper* es un bloque contenedor que permite "soltar" objetos en forma entidad o insertarlos en otros inventarios.
- El *dispenser* funciona igual pero si es posible los usa de distintas formas. P.e es capaz de disparar flechas.

4.1.2.6 Otros componentes

Por último hay raíles para vagonetas de distintos tipos que son utilizados frecuentemente por sus propiedades. En concreto el raíl propulsor y activador reaccionan a la *redstone* cambiando entre un modo activo e inactivo. En este caso se utiliza para propagar actualizaciones inmediatas debido a un error de diseño que permite mantenerlos encendidos sin recibir corriente.

4.1.3 Cancelación de actualizaciones

Debido a que las actualizaciones normalmente se producen de forma recursiva existe la posibilidad de desbordar la pila de llamadas de Java, causando una excepción (`StackOverflowError`). En la mayoría de los casos el juego acaba cerrándose ya que no sabe cómo recuperarse aunque, antes de terminar, guarda el progreso del mundo incluidas todas las actualizaciones previas a la excepción y, por tanto, las que quedan pendientes jamás llegan a ocurrir; son canceladas. Esta técnica se conoce como ***update suppression*** y se puede utilizar para conseguir estados normalmente imposibles. P.e. si cancelamos

la cadena de actualizaciones tras romper un bloque sobre el que hay una antorcha obtendríamos una antorcha que flota en el aire.

Convenientemente, si esta excepción se produce en la fase del jugador, la tarea correspondiente está rodeada por un bloque `try-catch` que imprime la excepción y continua con la ejecución de forma normal. Para evitar que el juego se cierre es importante que no haya ningún pistón involucrado en la cadena de actualizaciones ya que se retrasan hasta la fase de eventos como hemos explicado en su sección respectiva.

4.2 Chunks

El mundo de Minecraft está dividido en partes más pequeñas. Un *chunk* (del inglés, pedazo) es un grupo de $16 \times 16 \times 256$ bloques que el juego utiliza para diversas tareas como la generación procedural o la carga dinámica del mundo. Cada *chunk* está dividido verticalmente en 16 secciones, es decir, en grupos de $16 \times 16 \times 16$ bloques. Por otro lado, los *chunks* se guardan en disco en grupos de 32×32 llamados regiones.

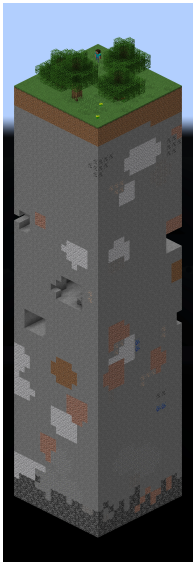

Figura 11: Chunk de 16x16x256 bloques.

4.2.1 Formato de mundo

El formato que utiliza Minecraft para guardar datos se llama NBT (*Named Binary Tag*). Es una estructura en forma de árbol donde cada elemento está definido por su tipo, un nombre y un valor. Esta estructura se serializa en binario y normalmente comprimido usando GZip.
Las regiones se almacenan en un formato llamado Anvil que consiste de una cabecera de tamaño estático compuesta por *offsets* a cada *chunk* que puede almacenar, seguida de los distintos *chunks* existentes en formato NBT comprimido. Si un offset es cero el *chunk* no está presente y viceversa. La estructura interna de un chunk se puede consultar en el Anexo I.

4.2.2 Savestate

El formato Anvil marca un límite de tamaño a cada *chunk* de ≈1MB (1044474 bytes). Si un *chunk* excede este tamaño ya comprimido, Minecraft no lo guarda en disco, por lo que la próxima vez que se cargue mantendrá el estado anterior. Esto es lo que comúnmente se llama *savestate* en referencia a los archivos homónimos usados por emuladores que habitualmente son un volcado completo del estado de la máquina y que permiten recuperarlo tal cual.

Aunque el espacio que ocupan los bloques es estático (si todas las secciones se han generado), podemos incrementar el tamaño del *chunk* usando tantas entidades y *tile entities* como sean necesarias. Se conocen varias formas de exceder el límite pero la más sencilla es usar libros, un objeto que te permite almacenar texto. Cada libro tiene un máximo de 50 páginas que permiten escribir varias líneas con una fuente de ancho variable. Debido a la compresión de GZip se necesitarían varios cientos de libros distintos completamente llenos para llegar a ese tamaño. En su lugar, el método habitual es utilizar libros llenos de caracteres únicos escogidos de forma aleatoria del rango de caracteres Unicode aceptado por el juego (Figura 12), el cual incluye, entre otros, más de 36 mil caracteres CJK (Chinese, Japanese and Korean). Estos libros son tan densos que menos de 60 copias de tan sólo dos libros distintos son más que suficiente como para que no se puedan comprimir de forma eficiente.

Aún así, estas copias necesitan colocarse de forma alterna en un inventario dentro del *chunk* o, en caso contrario, el *chunk* se guardará correctamente. Como el *savestate* revierte los cambios en el *chunk*, si introducimos los libros por primera vez en el *chunk*, estos se perderían, pero nos podemos aprovechar de lo que hemos mencionado para guardarlos "comprimidos" y alternarlos posteriormente sin perder las copias. Por encima, esto podemos automatizarlo usando *redstone* para realizar el proceso remotamente.

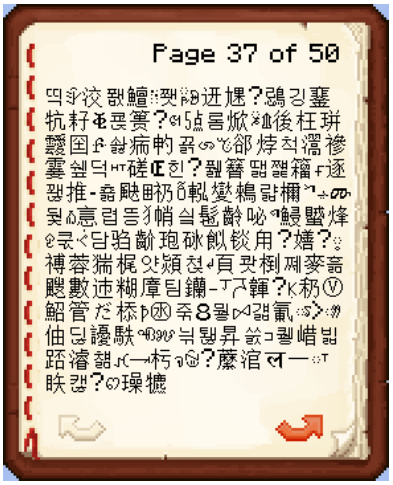

Figura 12: Una página de uno de los libros usados para hacer *save state*.

4.2.3 Carga

El juego no mantiene todo el mundo cargado en memoria, sólo aquellos *chunks* que estén lo suficientemente cerca del jugador y un área especial llamada *spawn chunks*. Este área tiene un tamaño de 16×16[4], está centrada en un lugar escogido aleatoriamente (spawn point) próximo a las coordenadas 0, 0 del mundo y se mantiene cargada permanentemente. El área que se carga alrededor de un jugador es un cuadrado variable dependiente de la distancia de dibujado configurada.

Además, otros elementos pueden llegar a cargar *chunks* temporalmente en cualquier fase del juego si se intenta acceder a ellos. Esto va desde una entidad haciendo *pathfinding* hasta una cadena de actualizaciones provocada por *redstone*, permitiéndonos cargar lo que queramos bajo demanda.

4.2.4 Generación y población

La primera vez que se intenta cargar un *chunk*, este se tiene que generar proceduralmente. Cuando se crea un mundo, se le asigna una semilla a partir de la cual se calculan múltiples generadores de ruido Perlin que se usan para crear la superficie del terreno, formaciones subterráneas y el mapa de biomas entre otras cosas. Muchos de los detalles, como la vegetación, los minerales y las estructuras[5], se posponen a una segunda fase de generación llamada **población**. Esta separación se debe a que la población se lleva a cabo en la esquina sureste del *chunk*, afectando a los otros 3 *chunks* colindantes. Por tanto, cada vez que se carga un *chunk*, este comprueba si cada una de las 4 áreas de 2×2 a las que pertenece están completamente cargadas (en la Figura 13 se puede ver que los *chunks* de abajo y la derecha no se han podido popular porque no están generados los chunks necesarios). Esto es así para reducir la posibilidad de que un elemento que se intente colocar cerca del límite del *chunk* se vea cortado. P.e. parte de la copa de un árbol podría no generarse si intenta acceder a un *chunk* que no esté cargado.

Como hemos visto anteriormente, el formato de *chunk* contiene una *flag* que indica si ya ha sido poblado o no. El cliente no dibuja los que tienen la *flag* desactivada para evitar que se vean aparecer los distintos elementos recién generados en la distancia[6]. Otra opción similar hubiera sido realizar

[4] El tamaño de los *spawn chunks* depende de dónde se encuentre el *spawn point*. Debido a que para pertenecer a los *spawn chunks* se tiene que cumplir la condición $-128 \leq (chunkCoord \cdot 16 + 8 - spawnpointCoord) \leq 128$ para las coordenadas x y z, el tamaño en cada eje será 17 en lugar de 16 si $spawnpointCoord = 8$.

[5] Algunas estructuras son creadas en la primera fase de generación pero se colocan en el mundo durante la población. Es principalmente el caso para aquellas que son muy modulares y ocupan múltiples *chunks*.

[6] Por cómo está planteado, en condiciones normales, los *chunks* del norte y oeste siempre tienen la *flag* activa a pesar de que sólo garantiza que su esquina sureste está poblada. Sin embargo el juego no respeta la distancia configurada ni es consistente a la hora de decidir qué dibujar, además de que hay niebla para ocultar dónde acaba el dibujado, haciendo prácticamente imposible de ver el efecto.

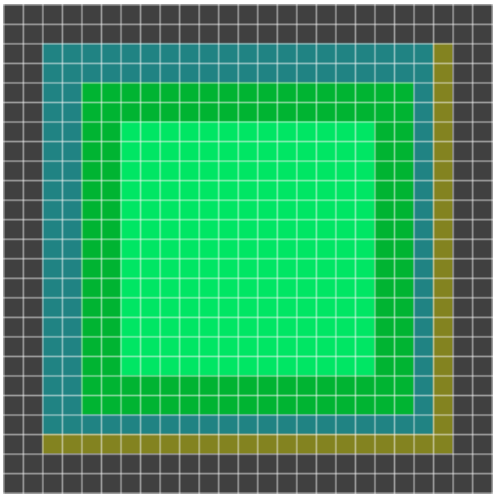

Figura 13: Representación de los diversos estados en los que se puede encontrar un *chunk*. Sólo los *chunks* verdes están cargados, los más oscuros son *lazy chunks*. Los *chunks* grises no están generados. Los *chunks* azules ya han realizado su población, los amarillos no.

la población centrada en el *chunk* y sólo si están cargados los 8 que lo rodean, pero a cambio no deberíamos dibujarlos para evitar el efecto anterior.

4.2.5 Descarga

Durante su propia fase, el juego intenta descargar los *chunks* almacenados en una estructura que contiene aquellos pendientes para ser descargados. Como hemos dicho antes, los *chunks* alrededor de los jugadores y los *spawn chunks* no pueden ser descargados y por lo tanto nunca acceden a esta estructura. Sin embargo, los que no cumplen con los requisitos no tienen porqué ser añadidos de forma inmediata, en su lugar son añadidos en los siguientes casos:

- **Movimiento del jugador**: a medida que el jugador se mueve por el mundo, aquellos que quedan fuera de su distancia de dibujado.
- **Auto-guardado**: cada 900 *ticks* (45 segundos) ocurre un auto-guardado en el que se itera por todos los *chunks* cargados.
- **Pausa**: en modo de un solo jugador ciertas interfaces pueden pausar el juego (el menú in-game, editar un cartel y abrir un libro) provocando el mismo efecto que el auto-guardado.

Una vez en la fase de descarga, se itera por la estructura y se procesa cada *chunk*, quitando cada uno de los elementos que contiene de las estructuras usadas para actualización, serializando toda su información y guardándola en disco. Pero antes de la descarga se comprueba si el *chunk* ha sido accedido desde que fue marcado para descargar, aunque no ocurre habitualmente debido a que entre ambas fases no ocurre ninguna de las principales. Además, para evitar que tome demasiado tiempo del tick, existe un límite arbitrario que impide descargar más de 100 *chunks* en el mismo tick, dejando el resto para los siguientes.

4.2.5.1 Permaloading

Abusando de los dos hechos anteriores podemos evitar que ciertos *chunks* se descarguen tras un auto-guardado. Si se descargan 100 *chunks* antes de los que queremos mantener cargados, estos "sobreviven" hasta el siguiente tick. Entonces, si podemos causar un acceso en cualquiera de las fases a lo largo de este, el juego ya no los descargará.

Causar la actualización es trivial, por ejemplo, un *hopper* presente en el *chunk* los causa todos los *ticks* en las condiciones adecuadas. Pero para garantizar que el *chunk* no es descargado en el primer tick primero necesitamos determinar el orden en el que se itera la estructura (`java.util.HashSet<T>`). Esta ocurre dependiendo del *hash* del elemento insertado, de menor a mayor, y aquellos que son iguales se mantienen en orden de inserción. Para más información, véase el Anexo II.

Por tanto, sólo necesitamos 100 *chunks* cuyo *hash* sea inferior al *chunk* que queremos mantener cargado, esto es lo que se conoce como un ***perma-loader***. También cabe mencionar que eso significa que no todos los *chunks* se pueden mantener cargados tan fácilmente pues, si deseamos mantener uno que tenga *hash* 0 también tendremos que tener en cuenta otras cosas como en qué orden son añadidos, pero sólo vamos a explorar el caso habitual por mantener la explicación simple.

Primero necesitamos saber cómo se calcula el hash de cada chunk. En primer lugar, en realidad, el `HashSet` no almacena chunks sino sus coordenadas x y z concatenadas en un `Long` tal que `long key = (x & 0xFFFFFFFFL) | (z & 0xFFFFFFFFL) << 32`. Luego, el cálculo del hash ocurre en la función `HashMap.hash` como podemos ver en el Código 1, que toma el `hashCode` que calcula el propio objeto, en este caso un `Long` y que podemos ver en el Código 2.

```
1  static final int hash(Object key) {
2      if (key == null) return 0;
3      int h = key.hashCode();
4      return h ^ (h >>> 16);
5  }
```

Código 1: Calculo del hash de un elemento en `java.util.HashMap`.

```
1  public static int hashCode(long value) {
2      return (int)(value ^ (value >>> 32));
3  }
```

Código 2: Calculo del hash de un objeto java.lang.Long.

Por último se aplica una máscara para evitar que exceda el tamaño del array subyacente como se puede ver en el Código 3. Esto no se suele tener en cuenta porque generalmente tiene un tamaño lo suficientemente grande para que los *chunks* con los que se trabajan no entren en conflicto ya que tienden a tener coordenadas bajas.

```
1  final V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) {
2      if (this.table == null || this.table.length == 0)
3          this.resize();
4      // table.length es siempre potencia de dos
5      int index = (table.length - 1) & hash;
6      Node<K,V> element = this.table[index];
7      if (element == null)
8          this.table[index] = new Node<K, V>(hash, key, value, null);
9      ...
10 }
```

Código 3: `java.util.HashMap.putVal`

En resumen, en las condiciones comunes el *hash* del *chunk* se puede simplificar a un `xor` entre ambas coordenadas. Esta operación es lo suficientemente sencilla para ver fácilmente que, siendo x y z iguales, el *hash* siempre será 0, por lo que es común construir el *perma-loader* en las "diagonales" del mundo. Hay que tener en cuenta un último detalle; el *perma-loader* es descargado en el primer tick tras el auto-guardado, por lo que es probable que queramos volver a cargarlo antes del siguiente. La forma general de conseguirlo es cargarlo desde un *chunk* con *hash* mayor.

4.2.6 Actualización: Lazy Chunks

Existen múltiples estructuras de datos que almacenan de diferentes formas los distintos elementos que se deben actualizar cada tick. Cuando un *chunk* es cargado, todo aquello que necesite ser actualizado es añadido a cada una de estas estructuras. Sin embargo en la comunidad existe el concepto de ***lazy chunk*** para indicar que no todos los *chunks* pertenecientes a un área de 5×5 *chunks* centrada en él están cargados (en la Figura 13 se pueden ver marcados los *lazy chunks*). Esto es así porque, durante la fase de entidades, por cada entidad presente en el mundo se comprueba si todos los *chunks* contenidos en el área[7] están cargados. En caso de que no lo estén, la entidad no es actualizada. Es decir, se llama *lazy chunks* a aquellos en los que las entidades no se procesan. Además, los bloques con gravedad, explicados en la Sección 4.1.1.2, también varían su comportamiento en estos *chunks* debido a cómo funcionan. Si se da el caso, en lugar de crear su entidad, simplemente itera verticalmente hacia abajo hasta encontrar un lugar donde colocar el bloque[8], por lo que "cae" instantáneamente. Cabe mencionar que uno de los bloques con gravedad, el huevo de dragón, funciona de forma distinta a los demás cuando se encuentra en *lazy chunks*. Por error, este se coloca un bloque más abajo, sobrescribiendo el que le hubiera servido de soporte, lo que nos permite eliminar bloques indestructibles fácilmente.

4.2.7 Manipulación de población

Podemos hacer que el juego repita la población de un *chunk* (incluso la generación completa) una y otra vez usando el *savestate* que vimos en la sección 4.2.2. El juego no guarda el *chunk* en el momento en que se genera, y aún menos tras la población, por lo que, si colocamos el *savestate* antes de que ocurra un auto-guardado, conseguiremos evitarlo. Por tanto, la próxima vez que se cargue el *chunk* tendrá que repetir esta acción.

Los elementos se colocan por lo general condicionalmente y durante la población todos usan la misma secuencia aleatoria. Si uno de ellos falla, la secuencia cambiará en cadena a partir de ese momento. Debido a que la población ocurre sobre 4 *chunks*, repetirla en un *chunk* significa que afecta a algunos que ya han pasado por el proceso. Como su terreno no es igual se generarán distintos elementos en distintas localizaciones. Por encima, estos *chunks* los podemos modificar libremente, por lo que efectivamente podemos manipular la población[9].

Dado que la semilla utilizada para popular el *chunk* es siempre la misma podemos predecir dónde se genera cada elemento, pero es común garantizar la consistencia de la secuencia aleatoria quitando todos los bloques posibles de los *chunks* afectados.

4.2.8 Cancelación de población

Durante la población se mantiene activa una *flag* global conocida como ***Instant Falling*** (IF), caída instantánea. Cuando está activa, los bloques con gravedad que vimos anteriormente pasan por alto la comprobación que vimos en la Sección 4.2.6 y se comportan como si la cumplieran, de ahí el nombre. A pesar de ello, está prácticamente en desuso porque la mayor parte de estos bloques se colocan en la primera fase de generación y no se actualizan. Sin embargo existen algunos elementos que causan actualizaciones durante la población, la mayoría son estructuras, haciendo caer estos bloques sobre ellas.

No obstante, hay otro elemento mucho más común que también produce actualizaciones. El juego intenta colocar múltiples bloques de agua y lava solitarias repartidas por todo el *chunk*. Estos pretenden ser fuentes que salen de formaciones rocosas para lo que el juego debe hacerlas fluir, de ahí que requieran

[7] Se comprueba el área comprendido por el tamaño de su *hitbox* + 32 bloques en cada dirección, por lo que si la entidad excediera los 16 bloques en alguna dirección podría ser un área mayor.

[8] Sin embargo la comprobación es distinta a la de la entidad y puede acabar colocándose sobre bloques que normalmente atravesaría, destruiría o que podrían destruir la entidad.

[9] Antes del descubrimiento del *savestate* ya se podía interactuar con la población usando mecanismos que nos permiten realizar modificaciones remotas, pero las interacciones posibles de estas técnicas son mucho más limitadas. Aún así se siguen utilizando porque el *savestate* no es adecuado para ciertos casos.

actualizarse. Sin embargo, los líquidos de Minecraft no fluyen inmediatamente, sino que programan la actualización de la misma forma que hemos visto en la Sección 4.1.1.2. Debido a esto, a la hora de poner cada uno de estos bloques se activa momentáneamente una segunda *flag* llamada ***Instant Tile Ticks*** (ITT). Mientras esta *flag* está activa, cualquier intento de programar una actualización la produce en el momento, permitiendo a los líquidos crear cascadas de forma inmediata.

Ahora, como vimos en la Sección 4.1.3, podemos cancelar actualizaciones llenando la pila de llamadas. Usando todo lo que hemos visto anteriormente podemos determinar dónde se generará uno de estos bloques y reaccionar a la actualización para cancelar la población a partir de este punto. Esto nos permite mantener ambas *flags* activas, permitiéndonos abusar de los compartimientos de ambas. Por ejemplo, ITT nos permite simplificar el proceso de cancelación usando *observers*, incluida la propia cancelación de población.

Cuando un *observer* es actualizado con ITT, este se activa inmediatamente y se desactiva recursivamente. Un segundo *observer* puede reaccionar a cada uno de estos cambios, pasando por el proceso de activarse y desactivarse dos veces. Si añadimos un tercero a la cadena se activaría cuatro veces, un cuarto ocho, y así sucesivamente con un crecimiento exponencial (2^n). Ahora, si creamos un bucle de *observers*, la última actualización acaba retro-alimentando el primer *observer*, incrementando la pila de llamadas lenta pero indefinidamente. En la Tabla 1 podemos ver el ejemplo más simple de esto.

Descripción	Etapa	Pila de llamadas
Inicialmente hay un observer A	◀	
Oponemos un segundo observer B	◀▶	▶
B se activa cuando se coloca	◀▶	▶▶
A es notificado y se activa	◀▶	▶▶◀
A se desactiva recursivamente	◀▶	▶▶◀◀
B se desactiva recursivamente	◀▶	▶▶▶
A es notificado y se activa	◀▶	▶▶▶◀
B es notificado y se activa	◀▶	▶▶▶◀▶
B se desactiva recursivamente	◀▶	▶▶▶◀▶▶
A se desactiva recursivamente	◀▶	▶▶▶◀◀
B es notificado y se activa	◀▶	▶▶▶◀◀▶
A es notificado y se activa	◀▶	▶▶▶◀◀▶◀
A se desactiva recursivamente	◀▶	▶▶▶◀◀▶◀◀
B se desactiva recursivamente	◀▶	▶▶▶◀◀▶▶
A es notificado y se activa	◀▶	▶▶▶◀◀▶▶◀
B es notificado y se activa	◀▶	▶▶▶◀◀▶▶◀▶
B se desactiva recursivamente	◀▶	▶▶▶◀◀▶▶◀▶▶
A se desactiva recursivamente	◀▶	▶▶▶◀◀▶▶◀◀
(Ad infinitum)	...	...

Tabla 1: Visualización de dos *observers* enfrentados llenando la pila de llamadas.

Hay un tercer efecto interesante que podemos causar cancelando la población. Como vimos anteriormente, cuando cargamos un *chunk*, este intenta iniciar la población, si es posible, de las cuatro áreas de 2×2 a las que pertenece. Esto ocurre en un orden concreto (sureste, suroeste, noreste y noroeste) por lo que, si al cargar un *chunk* dos de estás se completan y se cancela la primera, la segunda nunca llega a ocurrir. La diferencia de cancelarla de esta forma es que, al no comenzar el proceso de población, no se activa la *flag* que indica si el *chunk* ya está decorado. Por tanto, el *chunk* va a permanecer sin popular mientras se mantenga su área cargada, permitiendo al jugador acercarse e interactuar con él. En este estado se conoce como *invisible chunk* debido a que, como vimos en la Sección 4.2.4, el cliente no lo dibuja.

5 Threadstone

En esta sección vamos a ver cómo conseguimos invocar y abusar de un hilo de ejecución distinto al principal.

5.1 Beacon: análisis superficial

El ***beacon***, oficialmente en español "faro mágico", es un bloque con *tile entity* que cuenta con dos funciones especiales. El principal es que es capaz de aplicar diferentes efectos beneficiosos a los jugadores en un área de tamaño variable según sea configurado. En segundo lugar, como su nombre indica, funciona como un marcador emitiendo un rayo de luz vertical hacia arriba que permite ubicarlo en la distancia con facilidad. Además, para poder diferenciar distintos *beacons*, se puede cambiar el color del rayo colocando cristal tintado en su trayectoria (Figura 14).

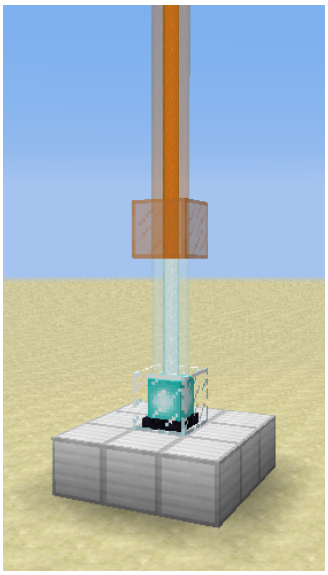

Figura 14: Un *beacon* activo cuyo rayo cambia de color al pasar por un bloque de cristal tintado.

```
1  public static void updateColorAsync(final World world, final BlockPos glassPos) {
2    HttpUtil.DOWNLOADER.submit(() -> {
3      Chunk chunk = world.getChunkFromBlock(glassPos);
4      for (int i = glassPos.getY(); i >= 0; --i) {
5        final BlockPos current = new BlockPos(glassPos.getX(), i, glassPos.getZ());
6        // Esta funcion utiliza el mapa de altura para comprobar si no hay
7        // un bloque impidiendo que el beacon emita el rayo.
8        // Sin embargo no se comprueba si el beacon lo emite de verdad.
9        if (!chunk.canSeeSky(current)) break;
10       if (world.getBlock(current) != BEACON) continue;
11       world.addSynchronizedTask(() -> {
12         TileEntity tileEntity = world.getTileEntity(current);
13         if (tileEntity instanceof TileEntityBeacon) {
14           tileEntity.updateBeacon();
15           world.addBlockEvent(current, BEACON, 1, 0);
16         }
17       }); } }); }
```

Código 4: `Beacon.updateColorAsync` reutiliza de forma cuestionable parte del código de red para invocar un hilo.

Lo que nos interesa de este bloque es precisamente esto último que, en realidad, ocurre en el sentido contrario. Independientemente de si hay un *beacon* o no, cada vez que se coloca un bloque de cristal tintado, se llama a la función que podemos ver en el Código 4.

Básicamente, el juego delega a otro hilo la tarea de buscar todos los *beacons* que haya debajo de él y, si por cada uno que encuentra, crea una tarea que se ejecutará en el hilo principal durante la fase de jugador, dónde finalmente se actualiza el *beacon*.

5.2 Beacon: problemas de la implementación

Aunque en principio esto pudiera parecer inofensivo, acceder a la misma memoria desde otro hilo puede causar inconsistencias. Esto de por sí no es un problema crítico porque sólo afecta al dibujado y, de hecho, este código sólo debería ejecutarse en el cliente pues el servidor no tiene nada que actualizar realmente. Por desgracia, aunque en estos casos es común observar al principio del método una comprobación como `if (world.isServer) return`, en este no ocurre, por lo que el servidor lo ejecuta de forma innecesaria.

Solicitar un chunk en el servidor tiene consecuencias mayores si de alguna forma no estuviera cargado. En ese caso, el juego no tendría más opción que cargarlo desde el hilo del *beacon*, lo cual puede ser problemático para la estabilidad del juego pues modifica varias estructuras que podrían estar siendo usadas por el hilo principal, causando potencialmente una excepción por modificación concurrente. Pero podemos llevarlo más allá si durante la carga ocurre una población pues, como vimos en la Sección 4.2.8, podemos conseguir actualizaciones durante esta fase, es decir, escrituras asíncronas. Por encima, haciendo más sencilla la ocurrencia, está implementado de forma que se accede al bloque a través del mundo en lugar del *chunk*, lo cual hace que el *chunk* se pida en cada iteración.

5.3 Cargando un chunk de forma asíncrona

En condiciones normales todo esto no debería ser posible, pues para crear el hilo es necesario colocar un bloque de cristal tintado y, aunque no estuviera cargado antes, el hecho de poner el cristal carga el *chunk*. A pesar de todo a día de hoy se conocen dos formas de conseguir esto:

- Manipulando la estructura dónde se almacenan los *chunks* cargados de forma que se produzca una condición de carrera que impida al juego encontrar el *chunk* que busca. Esto se conoce como *Chunk Swap*, en inglés “intercambio de chunk”.
- Retrasando la ejecución del hilo al menos un tick de forma que podemos descargar el *chunk* antes de que empiece la ejecución.

Cada uno tiene sus pros y sus contras, mientras que el último es más sencillo de ejecutar, es muy reciente y no está bien documentado aún, por lo que, en su lugar, vamos a explorar el *Chunk Swap*.

5.3.1 Manipulación del HashMap

Cada dimensión tiene su propia tabla *hash* dónde se almacenan sus *chunks* cargados, concretamente es la implementación de `Long2ObjectOpenHashMap` de la librería FastUtils 7.1.0. Como dice el nombre, a diferencia de la implementación de la JDK, la política de resolución de colisiones es de direccionamiento abierto. De hecho se resuelve de la forma más simple posible, si su espacio está ocupado se suma uno al índice hasta encontrar uno libre. El tamaño del array interno siempre es una potencia de dos y su política de redimensionamiento también es sencilla. Si al añadir un elemento este excede el factor de carga (75 %), este duplica su tamaño. Por el contrario, si al eliminarlo está por debajo del 25 % del factor (18,75 %), su capacidad se divide a la mitad. Esta implementación no está protegida contra accesos concurrentes, por lo que podemos manipularla por ambos hilos al mismo tiempo sin que lance una excepción. Por último, el *hash* se calcula usando la función de la misma librería `HashCommon.mix`, véase Anexo III, y el índice se obtiene con una máscara a partir del tamaño de la tabla. A diferencia

de la JDK, no hay un patrón sencillo en el que se organicen, así que se utilizan herramientas externas para calcular qué *chunks* tienen el *hash* que nos interesa.

Se conocen dos métodos de abusar de esta implementación.

- **Rehash method:** el primer método consistía abusar de que esta implementación tiene una entrada más en el array de claves para manejar el caso especial del *hash* 0 (chunk 0,0). Se basa en que, al final de un *rehash*, la máscara usada para calcular el índice en el array se actualiza antes que el array de claves. Si ocurre esto en el thread principal mientras el *beacon* thread intenta acceder a un *chunk* cuyo *hash* varía con la nueva máscara, normalmente *crashearía*. Sin embargo, si el *chunk* tiene un *hash* igual al tamaño del *hashmap*, al aplicar la nueva máscara coincidirá con la entrada extra y, si no está ocupada, entenderá que no estaba cargado. Este método requiere unas condiciones muy específicas por lo que ha quedado en desuso tras la aparición de estrategias con mayor probabilidad de éxito y flexibilidad.
- **Unload method:** es el más usado actualmente y consiste en, dado un *chunk* que queremos recargar, llamado habitualmente *glass chunk* porque es dónde se coloca el cristal tintado, cargar en orden:
  - Un *chunk* con igual *hash* que el *glass chunk*, conocido como *unload chunk*.
  - Luego n *chunks* que llenen las n siguientes entradas de la tabla, esto se conoce como *cluster*.
  - Por último el *glass chunk*, provocando que se desplace $n+1$ entradas de su posición original.

	hash		n		$n+1$	
...	Unload Chunk	...	Cluster	...	Glass Chunk	...

Tabla 2: Configuración de la tabla *hash* necesaria para el *chunk swap*.

De esta forma, si en el hilo principal descargamos el *unload chunk*, este tendrá que buscar si hay un *chunk* cargado que haya sido desplazado y ahora deba ocupar esa entrada, en este caso el *glass chunk*. Si mientras ocurre esto el thread secundario intenta obtener el *chunk*, este encontrará `null` en el índice correspondiente al *hash*, por lo que entenderá que no está cargado. A mayor *cluster*, más tarda el proceso de búsqueda y más fácil es que ocurra esta condición de carrera.

5.3.2 Sincronización de los hilos

Primero necesitamos asegurarnos que el hilo puede ejecutarse durante la fase de descarga. Observando el esquema del tick podemos ver que la fase más cercana a la descarga donde podemos trabajar es la de jugador. Debido a que la carga del hilo es mínima, probablemente termine el proceso demasiado pronto, por lo que tenemos que buscar formas de retrasarlo y reducir el tiempo entre fases.

En este último caso, sólo podemos reducir el tiempo de dos formas. En cada tick el juego intenta hacer aparecer enemigos y animales cerca de los jugadores en la fase de *mob spawn*. Sin embargo hay un límite de entidades que puede haber en el mundo de cada uno de los tipos de estos. Aunque no es necesariamente tan sencillo como "atraparlos", existen formas de llenar estos límites. Con esto, el tiempo de esta fase sería despreciable, pero el coste de esta ya es lo suficientemente bajo como para que sea obligatorio. En segundo lugar, en la misma fase de descarga podemos asegurarnos de que el primer *chunk* sea el que nos interesa. Esto puede ser una mejora importante pues la serialización de un *chunk* es un proceso bastante lento.

Ahora, para aumentar el ciclo de vida del hilo tenemos otras opciones más interesantes. La primera mejora consiste maximizar las iteraciones del bucle principal del hilo colocando el cristal en el bloque más alto posible y quitando todos los bloques que hayan debajo para que pueda seguir buscando. Además, si todos los bloques por los que pasa son *beacons* tendrá que ejecutar código extra. Esto no

va a incrementar drásticamente el trabajo de por sí, pero como veremos ahora sigue siendo interesante. Por otro lado, al encontrarse un *beacon* se ejecuta la función `addSynchronizedTask`. Esta está pensada para que los hilos encargados de las conexiones con los jugadores guarden las acciones para procesarlas en su fase correspondiente. El hecho de añadir a la cola de tareas pendientes se hace dentro de un bloque *synchronized* para evitar modificaciones concurrentes, por lo que sólo uno de los hilos puede ejecutarse mientras el resto esperan su turno.

```
1  public ListenableFuture<Object> addSynchronizedTask(Runnable runnable) {
2      Validate.notNull(runnable);
3      return this.<Object>callFromMainThread(Executors.callable(runnable));
4  }
5
6  public <V> ListenableFuture<V> callFromMainThread(Callable<V> callable) {
7      Validate.notNull(callable);
8
9      if (!this.isCallingFromMinecraftThread() && !this.isServerStopped()) {
10         ListenableFutureTask<V> task = ListenableFutureTask.<V>create(callable);
11
12         synchronized (this.futureTaskQueue) {
13             this.futureTaskQueue.add(task);
14             return task;
15         }
16     }
17     ...
18 }
```

Código 5: `addSynchronizedTask` tiene que esperar a que ningún otro hilo esté trabajando con la cola de tareas.

Esto supone, en primer lugar, que el hilo estará parado hasta que la fase de jugador no termine. Y segundo, si podemos ejecutar múltiples hilos poniendo más de un bloque de cristal tintado, estos se empezarán a bloquear mutuamente cada vez que encuentren un *beacon*.

Para ello necesitamos saber que podemos hacer todas las acciones que requiramos si provocamos una cadena de actualizaciones de *observers*. Como vimos en la Sección 4.2.8, el crecimiento de sus actualizaciones con ITT activo es exponencial, por lo que, aunque actualizar un *observer* tome 0,01ms, 20 encadenados superarían los 10s, 30 las 3 horas... Gracias a la separación cliente-servidor, el jugador puede seguir interactuando con el entorno a pesar de que el servidor esté ocupado, quedando todas sus acciones a la espera de ser procesadas. Esto nos permite, por ejemplo, colocar tantos bloques de cristal como sean necesarios de forma que se procesarán juntos durante la siguiente fase de jugador.

Con esto tenemos herramientas más que suficientes para llegar a la fase de descarga, ahora sólo necesitamos sincronizar la creación del *beacon* thread con la descarga del *unload chunk*. Como vimos en la Sección 4.2.5, para descargar un *chunk*, primero se marca y luego se descarga en su fase correspondiente. Es decir, tenemos que marcar el *unload chunk* de forma que este se descargue cuando nos interesa, para lo cual conocemos dos métodos:

Usar el movimiento del jugador es la que menos preparación requiere, pero necesita múltiples jugadores pues un jugador necesita estar en el *glass chunk* para iniciar el hilo. Generalmente el *unload chunk* suele estar lejos de este, por lo que es físicamente imposible que el mismo jugador haga ambas tareas[10]. Además, como veremos más adelante, nos interesa que el *unload chunk* sea el primero en ser descargado y la mejor forma de garantizarlo es sólo descargar ese *chunk*, cosa que se suele conseguir con un tercer jugador.

Por otro lado, nuestro primer aporte consiste en un método alternativo con un sólo jugador ya que, no sólo es conveniente para trabajar, pues no necesitamos múltiples instancias del juego abiertas;

[10]Mientras el juego está detenido en la fase de jugador tampoco se pueden cargar *chunks*, así que solo es posible si se encuentran ambos dentro de la distancia de dibujado.

sino que además lo hacemos accesible a los jugadores en modo un jugador. Aunque se puede adaptar usando mecanismos de sincronización con el auto-guardado, esta técnica se centra en abusar de la pausa tal y como detallamos en siguiente procedimiento:

Primero activamos un mecanismo capaz de actualizar dos *ticks* seguidos una cadena de *observers* lo suficientemente larga como para que el servidor se detenga unos segundos. Durante la primera cadena de actualizaciones, pausamos el juego simplemente. Una vez termina el servidor de procesar, comienza el segundo tick y, como el juego está pausado, marca todos los *chunks*, incluyendo el *unload chunk* y el *perma-loader* que necesitamos para mantener cargado el *cluster*. Tras reanudar el juego, el servidor descarga los 100 primeros *chunks* que constituyen el *perma-loader*, por lo que el *unload chunk* se queda pendiente para el siguiente *tick*. Después, durante este mismo tick, el mecanismo activa la cadena de *observers* una segunda vez. Mientras es procesada, el jugador coloca el cristal necesario para invocar los distintos hilos. Una vez termina la cadena de actualizaciones, empieza el siguiente tick en el cual, primero se procesan todos las acciones del jugador juntas, creando todos los hilos y, poco después, se descarga el *unload chunk*, creando la posibilidad de obtener un *chunk swap*.

5.3.3 Población asíncrona

La primera idea que se nos puede ocurrir es que, si el *glass chunk* era un *chunk* sin generar en el cual hemos colocado un *savestate*, al recargarse por el *Chunk Swap* se va a crear y popular de nuevo. Si bien es posible, mejorar las posibilidades de obtenerlo requiere colocar una gran cantidad de *beacons*, lo cual es lento y caro; y, como veremos más adelante, nos interesa tener la capacidad de repetirlo tantas veces como sea necesario. En su lugar, se suele crear un *chunk* invisible (véase Sección 4.2.8) adyacente al *glass chunk* de forma que inicie el proceso de población tras el *Chunk Swap*. En cualquiera de los dos casos necesitamos encontrar dónde se generan las fuentes de la misma forma que para la cancelación de población, pero esta vez no queremos reaccionar a la actualización causando una excepción. En su lugar podemos usar la cadena de *observers* que hemos visto anteriormente. Al igual que en el hilo principal, esto causa que el juego se detenga para procesar la cadena exponencial, en este caso mantenemos el proceso de población detenido mientras causamos tantas actualizaciones asíncronas como queramos.

5.4 Aplicaciones asíncronas

Ahora que tenemos disponibles las actualizaciones asíncronas vamos explorar cómo usarlas.

5.4.1 Falling Block Swap

Cuando los bloques con gravedad reciben una actualización, programan otra actualización un tick después. Para asegurarse de que el bloque no ha cambiado durante ese tick, se pasa el bloque la ha causado y si no es igual al actual se cancela. Sin embargo, tras la comprobación se vuelve a leer el bloque en esa posición por lo que, si cambiara entre esos dos puntos, crearía una entidad con ese bloque en lugar del original. Esto se conoce como ***Falling Block Swap*** y nos permite tener entidades de bloques que no se ven afectados por la gravedad. Además, si el bloque no puede colocarse dónde cae la entidad, esta se transforma en un objeto, permitiéndonos obtener objetos que no son posibles de obtener de forma normal.

Es debido a que no hay nada entre los dos puntos del código que se pueda manipular para conseguir esto que se empezó a investigar si era posible provocar que este código se ejecute en un hilo mientras otro hace el cambio. Una vez obtenemos actualizaciones asíncronas podemos propagarlas hasta el bloque con gravedad y dejar que el hilo principal se encargue de hacer el cambio. Aunque la posibilidad de que suceda la condición de carrera es muy baja de por sí, podemos mejorarla ampliando el tiempo entre la comprobación y la creación de la entidad. Para ello nos aprovechamos de la llamada a `isAreaLoaded` que ocurre entre ambas, que es de hecho la comprobación de Lazy Chunk que vimos en la Sección 4.2.6. Para hacer esto se tiene que acceder a la estructura dónde se almacenan los *chunks* cargados y, como vimos anteriormente, podemos hacer que este acceso sea más lento usando un

cluster. De hecho, podemos reusar fácilmente el *cluster* que hemos usado para el *Chunk Swap*. Como se comprueban 5×5 *chunks*, podemos incluso aprovechar que múltiples *chunks* estén desplazados de su posición original, aunque no nos interesa que todos los *chunks* tengan su acceso ralentizado. En `scheduleUpdate` también se hace una llamada a esta función, pero esta comprueba un área de 2×2 *chunks* centrada en el bloque para impedir que las actualizaciones instantáneas puedan cargar un *chunk* durante la población. Si esta llamada se ve afectada por el *cluster* la cadena de actualizaciones será más lenta, aumentando el tiempo que toma cada intento. Por tanto, se recomienda que sólo estén desplazados los *chunks* del anillo externo.

```
1  public void scheduleUpdate(BlockPos pos, Block block, int delay, int priority) {
2    if (this.instantTileTicks && block != AIR) {
3      if (block.requiresUpdates()) {
4        if (this.isAreaLoaded(pos.add(-8, -8, -8), pos.add(8, 8, 8))) {
5          // Lo lee, comprueba que no ha cambiado y llama a su funcion update
6          BlockState blockState = this.getBlockState(pos);
7          if (blockState.getBlock() == block)
8            blockState.getBlock().update(this, pos, blockState, this.random);
9        }
10       return;
11     }
12     delay = 1;
13   }
14   ...
15 }
16
17 public void update(World world, BlockPos pos, BlockState blockState, Random rng) {
18   if (!this.canFallThrough(world.getBlockState(pos.down())) || pos.getY() < 0)
19     return;
20   if (!this.instantFalling &&
21       world.isAreaLoaded(pos.add(-32, -32, -32), pos.add(32, 32, 32))) {
22     if (!world.isRemote)                          /* Lo vuelve a leer aqui */
23       world.spawn(new EntityFallingBlock(world, pos, world.getBlockState(pos)));
24     return;
25   }
26   ...
27 }
```

Código 6: Extracto de `World.scheduleUpdate` y `FallingBlock.update`.

5.4.2 Word Tearing

Como vimos, los *chunks* se dividen en hasta 16 secciones verticales que son las que de realmente contienen los bloques. Por tanto, las acciones de colocar u obtener un bloque se delegan a la sección correspondiente. Sin embargo, las secciones no guardan los bloques en un array mientras están en memoria, en su lugar utilizan una paleta que asigna a cada bloque un identificador y una estructura llamada `BitArray` que se encarga de almacenarlos. Con el objetivo de reducir la memoria necesaria, los identificadores que se usan tienen un número de bits o tamaño de palabra variable dependiendo de cuántos bloques distintos se encuentren en la sección[11]. También depende de esto que se uso uno de los siguientes tipos de paleta:

- **Linear:** 4 bits, hasta 16 bloques. La paleta contiene cada bloque en orden de aparición en un array interno y el identificador de cada uno es su índice. Buscar el identificador del bloque requiere iterar el array.

[11] El aire está siempre presente en la paleta aunque en la sección no lo esté.

- **HashMap:** 5-8 bits, de 17 hasta 256 bloques. Los identificadores se asignan de la misma forma, pero además se mantiene una tabla *hash* para reducir el coste de la búsqueda.
- **Registry:** 13 bits. Usa el identificador real del bloque, por lo que no es una paleta como tal.

Mientras el chunk está cargado sólo se puede aumentar el tamaño de la paleta, pero al descargar el chunk la paleta no se guarda, en su lugar se recalcula cuando al deserializar. Tampoco se pueden eliminar bloques de la paleta excepto que ocurra un redimensionamiento.

Por otro lado, `BitArray` es el encargado de concatenar los distintos bloques en un array de longs que cambia de tamaño junto a la paleta para poder almacenar todos los bloques de la sección. Resulta que, como se puede ver en el código 7, cuando la cantidad de bits por bloque que usa la paleta no es potencia de dos, el identificador del bloque se divide entre dos longs.

```
1  public void setAt(int index, int value) {
2      Validate.inclusiveBetween(0, this.arraySize - 1, index);
3      Validate.inclusiveBetween(0, this.maxEntryValue, value);
4      int absolute_bit = index * this.bitsPerEntry;
5      int start_long = absolute_bit / 64;
6      int end_long = ((index + 1) * this.bitsPerEntry - 1) / 64;
7      int relative_bit = absolute_bit % 64;
8      this.longArray[start_long] =
9          this.longArray[start_long] & ~(this.maxEntryValue << relative_bit) |
10         ((long)value & this.maxEntryValue) << relative_bit;
11
12     if (start_long != end_long) {
13         int cut_point = 64 - relative_bit;
14         int bits_left = this.bitsPerEntry - cut_point;
15         this.longArray[end_long] =
16             this.longArray[end_long] >>> bits_left << bits_left |
17             ((long)value & this.maxEntryValue) >> cut_point;
18     }
19 }
```

Código 7: `BitArray.setAt`.

Por tanto, si dos hilos están realizando actualizaciones en paralelo tal que, por ejemplo:

- La paleta es de 7 bits.
- Long 1 y long 2: dos longs consecutivos tal que long 1 guarda 3 bits y long 2 el resto.
- Hilo A: intenta poner 1111111 entre ambos longs.
- Hilo B: intenta modificar cualquier otro bloque en el long 2.

Es posible las actualizaciones se produzcan en el orden establecido en la tabla 3.

Pasos	Long2 Long1
Estado inicial	...00000000 00000000...
Hilo A lee y escribe long 1	...00000000 11100000...
Ambos hilos leen long 2	...00000000 11100000...
Hilo A escribe long 2	...00001111 11100000...
Hilo B escribe long 2	...00000000 11100000...

Tabla 3: Ejemplo de la condición de carrera necesaria para que ocurra *Word Tearing*.

Esta condición de carrera nos permite "mezclar" dos bloques en otro distinto, lo que se conoce en la comunidad como *Word Tearing*, aunque dependiendo del tipo de paleta podremos hacer cosas distintas:

- Al ser potencias de dos, las paletas de tipo *Linear* y *HashMap* de 8 bits no permiten *Word Tearing*, pero sí que puede que los hilos sobreescriban un cambio del otro, por lo que puede un bloque se duplique o desaparezca.

- En el resto de tamaños del tipo *HashMap* sí que es posible, permitiéndonos crear cualquier bloque presente en la paleta usando otros dos.

- Debido a que *Registry* usa el identificador real del bloque, las posibilidades que nos ofrece son mucho más amplias pues podemos crear cualquier bloque siempre y cuando seamos capaces de encontrar una combinación que nos lo permita. Esto incluye bloques que ni siquiera aparecen en el mundo porque están pensados para usos creativos como puede ser el bloque de comandos.

5.4.3 Generic Method

En la Sección 5.4.1 vimos cómo causar un *Falling Block Swap* en la teoría, pero no es tan sencillo en la práctica. El problema viene de que no podemos sustituir directamente un bloque con gravedad por otro bloque cualquiera; generalmente necesitamos quitarlo primero, por lo que quedaría aire hasta colocar el nuevo bloque, lo que lleva demasiado tiempo como para que ocurra aún con un *cluster* muy grande. El primer uso de *Falling Block Swap* se aprovechaba de que durante la población se puede sustituir el bloque directamente, sin embargo esto es un proceso lento y sólo se aplica a un pequeño grupo de bloques posibles.

En su lugar, el uso más habitual es en combinación con *Word Tearing*, lo que se conoce como *Generic Method* debido a que cubre la mayoría de los casos. Para ello usamos una paleta de tipo HashMap configurada de forma que al sustituir el bloque con gravedad por aire resulte en el bloque que deseamos obtener vía *Falling Block Swap*. Esto supone causar, no una, sino dos condiciones de carrera entre tres hilos en un orden específico, para lo cual necesitaremos haber realizado dos *Chunk Swaps*.

La figura 15 representa una línea temporal en la que, de izquierda a derecha, ocurren los distintos eventos necesarios para nuestro propósito. El hilo A representa el hilo principal que es será el encargado de crear las condiciones necesarias para ello, mientras los hilos B y C intentan provocar *Word Tearing* y *Falling Block Swap* respectivamente.

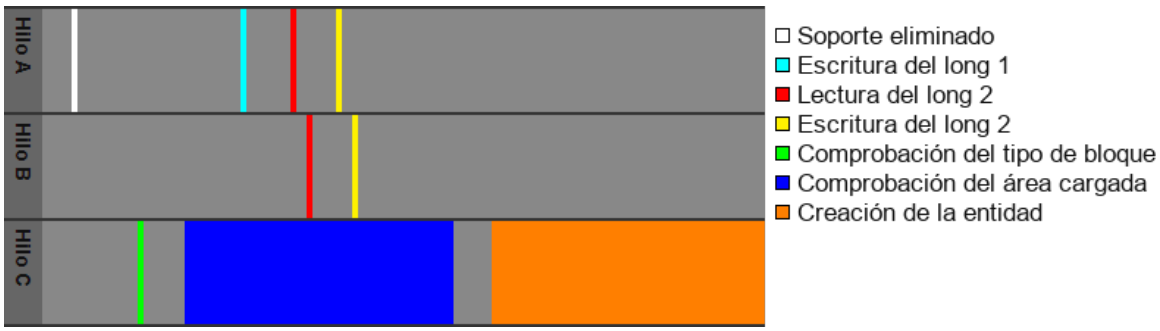

Figura 15: Línea temporal mostrando cómo deben solaparse los hilos para que ocurra *Generic Method*.

En resumen, el hilo A tiene que quitar el soporte al bloque con gravedad, permitiendo al hilo C que intente generar la entidad; y, rápidamente, mientras el hilo C está iterando el *cluster*, sustituir el bloque con gravedad por aire de forma sincronizada con el hilo B tal y como vimos en la Sección 5.4.2. Afortunadamente podemos realizar estas dos acciones rápidas retrayendo un pistón y *slime blocks* colocados de tal forma que el juego tenga que moverlos en el orden adecuado. Aún así, la posibilidad de que esto ocurra es muy baja por lo que conseguir una sola ocurrencia puede llevar un tiempo variable y elevado, por no hablar de los distintos problemas prácticos de los que no hemos hablado aún que hace que requiera atención constante.

6 Análisis de la Threadstone

Esta sección contiene nuestro análisis de este *glitch* para lo que entraremos en profundidad en algunos aspectos.

6.1 Cómo es posible Word Tearing

Observando el segmento de código en el ocurre *Word Tearing* se puede ver que entre la lectura y la escritura del array se ejecuta una cantidad de operaciones muy reducida. Esto hace que debiera ser casi imposible que esta condición de carrera suceda, sin embargo hemos observado que esta se produce con demasiada frecuencia. Como Java hace uso de un compilador JIT cabe la posibilidad de que las instrucciones fueran reordenadas, separándolas de forma significativa. Sin embargo, al inspeccionar el código desensamblado podemos ver que la distancia entre ambas es de a penas cinco instrucciones, haciendo aún más evidente la incógnita.

Esto es evidente cuando observamos el código de manera aislada y con el modelo de programación clásico donde una instrucción se ejecuta después de que se complete la anterior. Pero este modelo está obsoleto, es necesario tener en cuenta la ejecución especulativa de los procesadores fuera de orden.

En este caso, Minecraft, está diseñado para ser ejecutado en procesadores x86_64, en los que es típico tener ejecución especulativa y fuera de orden. En dichos procesadores, podemos distinguir, de manera muy simplificada, dos etapas: instrucciones en ejecución e instrucciones confirmadas. Aunque las instrucciones se puedan ejecutar fuera de orden, estas deben de ser confirmadas en orden [39]. Y es importante destacar, que en x86_64, hay ciertas instrucciones que no pueden terminadas de ser ejecutadas hasta que se confirman [40,41], para nuestro interés, las escrituras en memoria son de este tipo.

Volviendo al caso del *World Tearing*, teniendo en cuenta que cada uno de los bloques que realizan la lectura y escritura (Código 8), y que la estructura se encuentran precedida por más instrucciones (Anexo IV), permite que la ejecución especulativa llegue a realizar la lectura del dato de manera anticipada, aumentando el tiempo en la que la condición de carrera es posible hasta la escritura, ya que esta tiene que esperar a ser confirmada.

```
1  ; this.longArray[start_long] =
2  ;      this.longArray[start_long] & ~(this.maxEntryValue << relative_bit) |
3  ;      ((long)value & this.maxEntryValue) << relative_bit;
4    mov     r8, r9                    ; r8 = this.maxEntryValue
5    shl     r8, cl                    ; r8 = this.maxEntryValue << relative_bit
6    andn    r8, r8, qword ptr [rax + rsi * 8 + 18h] ; r8 = this.longArray[start_long] & ~r8
7    and     rbx, r9                   ; rbx = value & this.maxEntryValue
8    mov     r9, rbx                   ; r9 = rbx
9    shl     r9, cl                    ; r9 = (value & this.maxEntryValue) << relative_bit
10   or      r8, r9                    ; r8 = r8 | r9
11   mov     qword ptr [rax + rsi * 8 + 18h], r8     ; this.longArray[start_long] = r8
12   ...
13 ; this.longArray[end_long] =
14 ;   this.longArray[end_long] >>> bits_left << bits_left |
15 ;   ((long)value & this.maxEntryValue) >> cut_point;
16   mov     r9, qword ptr [rax + r11 * 8 + 18h]     ; r9 = this.longArray[end_long]
17   mov     ecx, r8d                  ; rcx = cut_point
18   sar     rbx, cl                   ; rbx = (value & this.maxEntryValue) >> cut_point
19   mov     ecx, edi                  ; rcx = bits_left
20   shr     r9, cl                    ; r9 >>>= bits_left
21   shl     r9, cl                    ; r9 <<= bits_left
22   or      r9, rbx                   ; r9 |= rbx
23   mov     qword ptr [rax + r11 * 8 + 18h], r9     ; this.longArray[end_long] = r9
```

Código 8: Extracto del método `BitArray.setAt` desensamblado dónde podemos ver ambas lecturas y escrituras.

6.2 Mejorando la estabilidad

El mayor problema que nos encontramos trabajando con múltiples hilos es que, al no estar pensado el juego para ello, ocurren condiciones de carrera inesperadas que nos impiden usarlos de forma prolongada. En este punto vamos a explorar algunos de los distintos problemas que surgen y las posibles soluciones que hemos encontrado para cada uno de ellos.

6.2.1 Finalización prematura del hilo

Uno de los problemas comunes es que el hilo termine antes de lo que debería, lo cual puede pasar fácilmente inadvertido porque, como veremos más adelante, el juego no se dibuja correctamente a menudo cuando trabajamos con actualizaciones asíncronas.

Debido a que no queda registro de ningún un error, la prueba que realizamos fue introducir todo el código ejecutado por el hilo en un bloque `try-catch` en el que se imprime la traza de ejecución y esperar a que ocurra mientras se hace uso normal de este. Específicamente se había observado mientras se intentaba *Generic Method* en el hilo encargado de hacer *Falling Block Swap*. Como era de esperar, eventualmente ocurrió permitiéndonos ver que en algún punto se lanza una excepción, concretamente `NullPointerException`, al intentar obtener una *tile entity*.

```
1  // Caused by: java.lang.NullPointerException
2  //    at net.minecraft.world.World.getPendingTileEntity(World.java:2413)
3  //    at net.minecraft.world.World.getTileEntity(World.java:2398)
4  //    at net.minecraft.block.BlockPistonMoving.neighborChanged(BlockPistonMoving.java:172)
5  //    at net.minecraft.world.World.notifyBlock(World.java:582)
6  //    at net.minecraft.world.World.notifyNeighborsOfStateExcept(World.java:550)
7  //   ...
8
9  private TileEntity getPendingTileEntity(BlockPos position) {
10     for (int i = 0; i < this.addedTileEntityList.size(); ++i) {
11         TileEntity tileEntity = this.addedTileEntityList.get(i);
12         if (tileEntity.isValid() && tileEntity.getPosition().equals(position))
13             return tileEntity;
14     }
15     return null;
16 }
17
18 public TileEntity getTileEntity(BlockPos position) {
19   if (this.isOutsideBuildHeight(position)) return null;
20   TileEntity tileEntity = null;
21   if (this.updatingTileEntities)
22     tileEntity = this.getPendingTileEntity(position);
23   if (tileEntity == null)
24     tileEntity = this.getChunkFromBlockCoords(position)
25                       .getTileEntity(position, CreationPolicy.IMMEDIATE);
26   if (tileEntity == null)
27     tileEntity = this.getPendingTileEntity(position); // <-
28   return tileEntity;
29 }
30
31 public void neighborChanged(BlockState state, World world, BlockPos position,
32                             Block neighborBlock, BlockPos neighborPosition) {
33     if (!world.isRemote)
34         world.getTileEntity(position);
35 }
```

Código 9: Traza del *crash* que estamos tratando y funciones relacionadas.

En primer lugar necesitamos entender porqué se busca una *tile entity*. El pistón en realidad hace uso de tres bloques distintos, la base del pistón que maneja la activación y desactivación de este; el brazo del pistón, que es un bloque principalmente visual; y el bloque desplazándose que es el encargado de hacer la animación de movimiento y colocar los bloques al final de esta. Este último bloque tiene su propia *tile entity* para almacenar el bloque que representa y el progreso de la animación. Cuando este recibe una actualización obtiene su *tile entity* pero nunca opera con ella con el único efecto posible de que esta se cree si no existe ya, cosa que no es posible en condiciones normales y la única forma "natural" de que ocurra en otro hilo es inocua[12].

Viendo el código de `World.getTileEntity` podemos ver que no ha podido obtenerla del `Chunk.getTileEntity`. Esta función devuelve y, gracias al segundo parámetro, crea si no existe la entidad para el bloque en la posición dada. Para que la ejecución continúe por `World.getPendingTileEntity`, la función tiene que devolver `null`, por lo que ni existe la *tile entity* ni el bloque que ha encontrado es un bloque desplazándose. Por tanto, el problema ocurre, como mínimo, durante la fase de eliminación de *tile entities* en adelante.

Por otro lado, `World.getPendingTileEntity` sólo itera la lista de entidades que están pendientes de ser añadidas. Durante la fase de actualización no se pueden añadir *tile entities* directamente al mundo, en su lugar se colocan en una lista donde quedan pendientes hasta terminar la actualización de estas. Por ejemplo, el *perma-loader* suele estar compuesto por *hoppers* que cargan los *chunks* que lo componen. Los *hoppers* son *tile entities* y se actualizan en esta fase, por lo que, si carga un *chunk*, este tiene que colocar las *tile entities* que contenga en la lista en lugar de añadirlas directamente.

Finalmente, el problema ocurre al intentar acceder a un miembro de la *tile entity* extraída que resulta ser `null`, cosa que no debería ser posible ya que nunca se ha añadido a la lista. El motivo por el que ocurre esto es que la llamada a `ArrayList.get` ocurre justo cuando el hilo principal ejecutando `ArrayList.clear` al terminar de procesar las *tile entities* pendientes. Como se puede ver en el Código 10, debido a que la JVM usa un recolector de basura para manejar la memoria, la lista debe eliminar todas las referencias de la lista sustituyéndolas por `null` antes de "reducir" su tamaño. Esto permite que se pase la comprobación de rango y obtenga `null` de ella.

```
1  public void clear() {
2      modCount++;
3
4      // clear to let GC do its work
5      for (int i = 0; i < size; i++)
6          elementData[i] = null;
7
8      size = 0;
9  }
10
11 public E get(int index) {
12     rangeCheck(index);         // if (index >= size)
13                                //    throw IndexOutOfBoundsException(OoBMsg(index));
14     return elementData(index); // return (E) elementData[index];
15 }
```

Código 10: Métodos relevantes de la implementación de `ArrayList` de JDK 8.

La forma más sencilla de evitar esto es evitando obtener `tile entities` desde cualquier hilo que no sea el principal. La solución que es utilizada actualmente consiste en utilizar una "barrera", usando raíles que podemos activar y desactivar desde el hilo principal, de tal forma que impedimos que la cadena de *observers* cause actualizaciones asíncronas mientras no nos sea interesante[13].

[12] Es posible que intentar añadir una *tile entity* que requiere actualización, como es el caso, justo antes de que empiece la fase de actualización de estas, de tal forma que se añada a la lista una vez que ya se está iterando, pueda causar una excepción de modificación concurrente. Sin embargo los pistones se procesan siempre en la fase de eventos y tanto el bloque como la *tile entity* son colocados en este momento.

[13] Para evitar que los hilos interactúen entre sí nos aprovechamos de cómo se propaga la corriente por los raíles

Por encima, esta técnica nos permite solucionar un segundo problema del que no hemos hablado hasta ahora. Si aplicamos *Word Tearing* tal cual lo hemos explicado existe más de un resultado para la misma operación, lo que se conoce como un subproducto. Por ejemplo, tomando la combinación mostrada en la figura 3, si falla tenemos que volver a colocar el bloque en posición para reintentarlo. Con el hilo aún causando escrituras, en el momento que lo colocamos de nuevo puede ocurrir la combinación contraria, creando el bloque "1111000" reemplazando del bloque "1111111" original. Teniendo ambos casos la misma probabilidad de ocurrir esto supone una pérdida importante de recursos y eficiencia pues requiere detectar y reparar el sistema, algo habitualmente manual debido a limitaciones. Por encima, como hemos visto, para hacer *Generic Method* usamos un pistón por lo que temporalmente habrá un bloque moviéndose, incrementando el número de posibles subproductos y la probabilidad de fallo. Usando una segunda barrera en la cadena de *observers* encargada de hacer *Word Tearing* podemos evitar todos los subproductos y pérdidas de materiales, permitiendo que funcione de forma autónoma.

6.2.2 Crash por desincronización

En otras ocasiones, es el juego el que *crashea* de forma aparentemente aleatoria. A diferencia del anterior caso, al ocurrir en el hilo principal, este sí que imprime en el registro la traza. En ella podemos ver que ocurre durante la fase del *PlayerChunkMap*, llamada así porque ocurre enteramente en una llamada a `PlayerChunkMap.tick`. Esta clase que se encarga de gestionar los *chunks* que están en la distancia de dibujado de al menos un jugador para generarlos si es necesario y mandar los cambios a los clientes. La excepción (`ConcurrentModificationException`) ocurre en este último caso, mientras se itera un `HashSet` idéntico al que ya vimos que se utiliza en la fase de descarga. Este contiene una entrada (`PlayerChunkMapEntry`) por cada *chunk* con información sobre los cambios que han ocurrido.

```
1 public void tick() {
2   ...
3   if (!this.entriesToUpdate.isEmpty()) {
4     for (PlayerChunkMapEntry entry : this.entriesToUpdate)
5       entry.update();
6     this.entriesToUpdate.clear();
7   }
8   ...
9 }
```

Código 11: `PlayerChunkMap.tick`.

En principio, el motivo del *crash* parece simple, por tanto. Si un *chunk* que no había sido modificado durante un tick es modificado por la línea asíncrona durante esta fase, se lanza la excepción. La solución por tanto es tan sencilla como asegurarse de que todos los ticks ocurre una modificación en el hilo principal para asegurarse de que existe la entrada. Sin embargo, este es sólo un posible motivo de que esto ocurra y de entre ellos el menos común pues la cadena de *observers* pasa la mayor parte del tiempo actualizando los últimos *chunks* por los que pasa, por lo que prácticamente garantiza que hayan sido modificados antes de esta fase. Aún así es importante ya que es habitual ver este caso cuando se crea el hilo y se propagan las actualizaciones por primera vez, provocando un *crash* justo después de un *chunk swap*.

Curiosamente, tras aplicar la solución anterior, la excepción sigue ocurriendo en el mismo lugar, por lo que por algún motivo la entrada no es añadida a la colección. Esto tiene sentido si lo relacionamos con que, uno de los efectos que caracterizan este *exploit* es que aquellos *chunks* en los que ocurren actualizaciones asíncronas tienden a dejar de hacerlo visualmente.

potenciadores o activadores. Sin entrar en detalle, a diferencia de la *redstone* no hay niveles y la propagación ocurre sólo cuando un raíl cambia de estado, por lo que le damos corriente de forma que el bloque con el que la cadena de *observers* debería interactuar no se actualiza pero debería estar activado.

Cuando ocurre un cambio en un *chunk*, el juego pide la entrada correspondiente y, si existe, se le notifica que ha ocurrido un cambio en esa posición. Sin embargo, sólo se añade la entrada a la colección si el número de cambios era cero anteriormente.

```
1  public void blockChanged(int x, int y, int z) {
2    if (!this.sendToPlayers) return;
3    if (this.changes == 0)
4      this.playerChunkMap.addEntry(this);
5    this.changedSections |= 1 << (y >> 4);
6    if (this.changes < 64) {
7      short index = (short)(x << 12 | z << 8 | y);
8      for (int i = 0; i < this.changes; ++i)
9        if (this.changedBlocks[i] == index) return;
10     this.changedBlocks[this.changes++] = index;
11   }
12 }
```

Código 12: `PlayerChunkMapEntry.blockChanged`.

Como vimos antes, hasta que no termina de iterar por todas las entradas a actualizar, el juego no limpia la colección dónde están almacenadas. Por tanto, si durante este tiempo el *chunk* es actualizado, el número de cambios se incrementará y, como ya está presente en el mapa, una vez se limpie la lista quedará excluido. El *chunk* estaría por tanto permanentemente desincronizado con el cliente si no fuera por un bloque de código adicional presente al principio de `PlayerChunkMap.tick`.

```
1  public void tick() {
2    long time = this.world.getTotalWorldTime();
3    if (time - this.lastUpdate > 8000L) {
4      this.lastUpdate = time;
5      for (int i = 0; i < this.playerEntries.size(); ++i)
6          this.playerEntries.get(i).update();
7    }
8    if (!this.entriesToUpdate.isEmpty()) {
9    ...
10 }
```

Código 13: `PlayerChunkMap.tick`.

Cada 8001 ticks[14] el juego itera por todos los *chunks* cargados por los jugadores y los actualiza independientemente de si han habido cambios o no. Esto significa que los *chunks* excluidos son actualizados y su contador de cambios es reiniciado y, por tanto, puede reinsertarse en la colección aunque sólo sea por un breve periodo de tiempo. Por desgracia, a pesar de que es lógicamente innecesario, se sigue pasando por el bloque anterior en lugar de simplemente limpiar la lista. Como consecuencia cabe la posibilidad de que vuelva a ocurrir el mismo *crash*, pero esta vez no tenemos forma de prevenirlo pre-insertando las entradas.

La única opción que tenemos por tanto es evitar que se produzcan actualizaciones asíncronas mientras ocurre esto, por lo que necesitaremos un temporizador sincronizado con este evento. La parte práctica de estas soluciones no es muy interesantes para este documento, pero vamos a ver las diferentes alternativas teóricas disponibles. Una idea consiste en mantener ocupado el hilo en un proceso lo suficientemente largo como para que pudiera pasar al menos un tick sin producir actualizaciones, pero por desgracia no conocemos a día de hoy una forma práctica de hacer esto. Por otro lado, en lugar de detener las actualizaciones, podemos dejar que ocurran de forma que no puedan actualizar una entrada. Una manera de conseguir esto sería dejar el mecanismo funcionando sin que ningún jugador

[14] Probablemente esto no es intencional y debiera ser cada 8000 ticks. Este es un tipo de error de programación muy común conocido como *off-by-one error*.

esté cerca. Al estar el *chunk* fuera de la distancia de dibujado, no debería haber una entrada para este , aunque existiera, nunca se reiniciaría. Esto elimina la necesidad del temporizador que hemos mencionado, sin embargo, por lo general es preferible estar cerca para poder ver que todo funciona correctamente y modificar cosas si fuera necesario. Debido a esto, la opción que se suele utilizar consiste en redirigir temporalmente las cadenas de *observers* a *chunks* fuera de la distancia de dibujado. Para ello necesitamos hacer uso de múltiples barreras de las que hemos mencionado en la sección anterior para bifurcar la cadena sin que los hilos interfieran entre sí. Además, al alejarlo vamos a provocar actualizaciones en bastantes *chunks* intermedios potencialmente causando la variación del *crash* que hemos tratado anteriormente, por lo que, hay que aplicar la misma solución en todo el trayecto. Una última opción curiosa, en caso de no querer cargar *chunks* adicionales, se puede redirigir en su lugar a un *chunk* invisible ya que, mientras no haga su población, nunca se envían los cambios al cliente.

6.2.3 Superposición de Chunk Swaps

Usando el método que hemos visto para hacer un Chunk Swap podemos ver a menudo que el juego *crashea* por una excepción al intentar actualizar una *tile entity*. Si bien este problema se suele ignorar porque es más sencillo volver a intentarlo por fuerza bruta hasta que funcione, sería conveniente ver qué está ocurriendo y prevenirlo si es posible.

Analizando la traza podemos ver que el problema ocurre durante la fase de actualización de *tile entities*. Mientras se itera la lista de *tile entities* el juego lanza `NullPointerException` al intentar acceder a una de ellas. Este problema es muy similar a lo visto en la Sección 6.2.1, pero esta vez es el caso inverso, pues no hay forma de que en un hilo que no sea el principal se eliminen elementos de la lista, aún menos que se limpie entera. Esto significa que, de alguna forma, `null` está presente en la lista en este momento.

Se ha observado que es posible que más de un hilo cargue el *glass chunk* durante el mismo *chunk swap*. Si esto ocurre, ambos tienen que introducir en la lista todas los *tile entities* presentes en este. Viendo el Código 14, en el caso de que las ejecuciones de dos hilos se solapen al insertar un elemento tal que:

Hilo A: `elementData[size] = e`

Hilo B: `elementData[size] = e`

Hilo A/B: `++size`

B sobrescribe la *tile entity* de A dejando que el último elemento de la lista sea `null`. Cabe destacar que esta condición de carrera es aparentemente aún más complicada que la de *Word Tearing*, sin embargo es posible por el mismo motivo, pues la llamada a `ensureCapacityInternal` realiza al menos una comprobación y puede incluso causar un redimensionamiento del array interno.

```
1 public boolean add(E e) {
2     ensureCapacityInternal(size + 1);
3     elementData[size++] = e;
4     return true;
5 }
```

Código 14: Código del método `java.util.ArrayList.add`.

Resolver este *crash* no es simple, pues necesitamos asegurarnos de que un sólo hilo intente crear el *chunk*. Podemos reducir la probabilidad de que ocurra reduciendo el tamaño del *cluster* para que el hilo principal tarde menos en ocupar el hueco, pero esto es un compromiso ya que, con una ventana más pequeña, el *Chunk Swap* será también menos probable. Si quisiéramos solucionarlo del todo tendríamos que usar un sólo hilo dentro del *glass chunk*, pero podemos seguir utilizando otros hilos en otro *chunk* para ralentizar este. A cambio, sólo un hilo puede hacer el *Chunk Swap*, por lo que necesitaremos un *cluster* mucho más grande para que coincida. El nuevo método para conseguir un *Chunk Swap* usa este concepto de forma no intencional y no requiere de un *cluster* por lo que no tiene este problema.

7 Conclusiones y Trabajo Futuro

A lo largo de la investigación hemos visto que, para un jugador casual, los *bugs* pueden tener un impacto negativo, sin embargo los *exploits* tienden a pasar completamente desapercibidos o simplemente se convierten en una anécdota. A cambio hemos visto que muchos jugadores han hecho de ellos un pasatiempo y para algunos es incluso una parte importante de su trabajo, ya sean jugadores profesionales, creadores de contenido o periodistas; mientras que, para los desarrolladores, puede suponer que su juego reciba más atención de la esperada y tenga una vida prolongada.

Sin embargo, no hay que dejar de lado el papel educativo de los mismos. Para llegar a comprender, explicar, o incluso seguir explorándolos a menudo se requieren conocimientos muy diversos. Desde estructuras de datos simples, hasta grandes y complejos algoritmos que, además, en la mayoría de los casos se han de obtener por ingeniería inversa y que a menudo lleva a la creación de diversas herramientas para analizar y depurar el juego. En muchos casos, la estadística juega un papel importante en el análisis de la situación. Todos estos valores son importantes, ya que demuestran muchas habilidades para aquellas personas que se encuentran dentro de este entorno. Siendo un punto de partida para que algunos acaben dentro de la industria de los videojuegos, o incluso en ramas relacionadas (seguridad, optimización, sistemas embebidos).

En ocasiones podemos ver que los jugadores toman el rol de divulgadores, dando lugar a que el público general adquiera conocimientos sobre informática y diseño de videojuegos que, aunque sea a nivel superficial, los familiariza con el entorno e incluso puede llegar a despertar su curiosidad en esta materia.

En el caso que hemos estudiado, podemos ver que hemos necesitado conocimiento previo de varias ramas en informática para comprender superficialmente cómo adaptar y ejecutar el *exploit*. Desde ingeniería inversa para obtener tanto el código del juego, como para su interpretación pasando por el *bytecode* intermedio de Java y el ensamblador de la máquina, hasta comprensión de algoritmia y paralelismo en un caso de uso real. Muchas de estas competencias no son fácilmente desarrollables en entornos de "juguete".

Es la primera vez que se explota la ejecución paralela en un videojuego a esta escala y probablemente sea la última. Es capaz de mostrar cómo un conjunto de pequeños detalles que individualmente no tienen gran impacto, pueden producir un cambio final muy grande. Aunque sienta un precedente para este tipo de *exploit*, probablemente sea más una instancia curiosa e irrepetible, por lo que resulta muy interesante para estudiar, comprender, explorar y enseñar.

En lo que al desarrollo de este *exploit* se refiere, todavía falta refinamiento y documentación adecuada para los distintos descubrimientos que han ocurrido a su alrededor, incluidos algunos usos menores y muchos detalles de los que no hemos hablado. Es probable que aparezcan nuevos usos, mejoras en los distintos procedimientos o nuevas estrategias directamente. De hecho, esta investigación se ha realizado en la versión 1.12 de Minecraft, pero hay investigaciones activas para otras versiones, tanto inferiores como superiores. Debido a los cambios, en algunas versiones es imposible que ocurra, en otras se ha conseguido parcialmente y en algunas es demasiado inestable como para ser usado prácticamente.

Por otro lado, en lo que a videojuegos futuros se refiere, los *exploits* van a seguir siendo relevantes en sus distintas comunidades. Aunque hay cierta tendencia a que sean errores menores gracias a la protección del sistema y el uso de motores que simplifican los trabajos más complejos, el conocimiento de los jugadores y las herramientas que se utilizan también siguen avanzando gracias a la popularidad que ha ido adquiriendo este tipo de comunidades con el tiempo.

Referencias

[1] Software bug - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Software_bug. [Accessed 30-May-2023].

[2] Emergent gameplay - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Emergent_gameplay. [Accessed 30-May-2023].

[3] Página web de Twin Galaxies. https://www.twingalaxies.com/, 2023.

[4] Games Done Quick Event Tracker. https://gamesdonequick.com/tracker/, 2023.

[5] Tool-assisted speedrun - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Tool-assisted_speedrun. [Accessed 30-May-2023].

[6] Scott "Pannenkoek2012" Buchanan. Super Mario 64 - A Button Challenge (Playlist). https://www.youtube.com/watch?v=e_IEhqUAuCY&list=PLmvvhlevqC2d6vis077yFHuLJHYtTtqe_, 2012 - 2023.

[7] Bismuth. The Complete History of the A Button Challenge. https://www.youtube.com/watch?v=yXbJe-rUNP8, 2022.

[8] Halopedia - Sword flying (Feasibility and version differences). https://www.halopedia.org/Sword_flying#Feasibility_and_version_differences, 2018.

[9] Jeremy Winslow. Minecraft Reached 140 Million Monthly Users And Generated Over $350 Million To Date. https://www.gamespot.com/articles/minecraft-reached-140-million-monthly-users-and-generated-over-350-million-to-date/1100-6490962/, 2021.

[10] Sandbox game - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Sandbox_game. [Accessed 30-May-2023].

[11] EthosLab. Canal de YouTube de EthosLab. https://www.youtube.com/@EthosLab, 2008 - 2023.

[12] Minecraft Discontinued Features Wiki. https://mcdiscontinued.miraheze.org/wiki/Main_Page, 2021.

[13] Retro Gamer. Nishikado-San Speaks. Retro Gamer. No. 3. p. 35., 2004.

[14] Edge Staff. The Making Of: Asteroids. https://web.archive.org/web/20121031020857/http://www.edge-online.com/features/making-asteroids/, 2009.

[15] Bismuth. How is this speedrun possible? Super Mario Bros. World Record Explained. https://youtu.be/_FQJEzJ_cQw?t=336, 2018.

[16] Jason Schreier. Quality Assured: What It's Really Like To Test Games For A Living. https://kotaku.com/quality-assured-what-it-s-really-like-to-play-games-fo-1720053842, 2017.

[17] Retro Game Mechanics Explained. Super Mario Bros. Glitch Levels Explained. https://www.youtube.com/watch?v=1ysdUajrhL8, 2022.

[18] Edge Staff. Pokémon Figures - The Pokémon Company. https://web.archive.org/web/20171122142832/http://www.pokemon.co.jp/corporate/en/data/, 2017.

[19] MissingNo. https://glitchcity.wiki/wiki/MissingNo. [Accessed 30-May-2023].

[20] luckytyphlosion. Dry underflow. https://archives.glitchcity.info/forums/board-107/thread-7175/page-0.html. [Accessed 30-May-2023].

[21] Expanded item pack. https://glitchcity.wiki/wiki/Expanded_item_pack. [Accessed 30-May-2023].

[22] Arbitrary code execution. https://glitchcity.wiki/wiki/Arbitrary_code_execution. [Accessed 30-May-2023].

[23] Speedrun.com - Games. https://www.speedrun.com/games. [Accessed 30-May-2023].

[24] Backward Long Jump - Ukikipedia. https://ukikipedia.net/wiki/Backwards_Long_Jump. [Accessed 30-May-2023].

[25] n64decomp. Super Mario 64 (Decompilation) - GitHub. https://github.com/n64decomp/sm64, 2019. [Accessed 30-May-2023].

[26] pannenkoek2012. SM64 - The Science of Cloning. https://www.youtube.com/watch?v=9xE2otZ-9os, 2014.

[27] TheRedKorsar. portal chamber 18 least time 0:09. https://www.youtube.com/watch?v=dnddnwAWHFs, 2010.

[28] Martin Szeibert. Portal 2 - Turret Factory OOB Route. https://www.youtube.com/watch?v=QdRcAmOrvGQ, 2015.

[29] Aero. How to CLONE INFINITE ITEMS & LEGENDARIES GLITCH in Pokemon Scarlet & Violet. https://www.youtube.com/watch?v=wJwD-d1HMtE, 2022.

[30] alyo. Paraglider Duplication Glitch explained in under 2 minutes - Legend of Zelda: Tears of the Kingdom. https://www.youtube.com/watch?v=faQhuHWi9j4, 2023.

[31] Joseph "Earthcomputer" Burton. [1.12] The Quest for the Bedrock Item. https://www.youtube.com/watch?v=l988pdLw8O4, 2020.

[32] Joseph "Earthcomputer" Burton. [1.12] How to get the Bedrock Item. https://www.youtube.com/watch?v=YHdSpO-Gsvc, 2020.

[33] coolmann24. It works. https://www.youtube.com/watch?v=TiQMMwMJIzM, 2020.

[34] Hans-Juergen Boehm. How to Miscompile Programs with "Benign" Data Races. In *HotPar*, pages 3–3, 2011.

[35] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective Data-Race Detection for the Kernel. In *OSDI*, volume 10, pages 1–16, 2010.

[36] et al. Myren Eario, Earthcomputer. Falling block playlist. https://www.youtube.com/watch?v=BQnejuEjMJs&list=PL8r-bvM9ltXOCEQMW_WTvQWUfmwVl528h, 2022.

[37] Obfuscation map - minecraft wiki. https://minecraft.fandom.com/wiki/Obfuscation_map. [Accessed 30-May-2023].

[38] et al. Michael "Searge" Stoyke, Thomas "ProfMobius" Guimbretière. Mod coder pack. http://www.modcoderpack.com. [Accessed 30-May-2023].

[39] James E Smith and Gurindar S Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609–1624, 1995.

[40] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22*, pages 391–407. Springer, 2009.

[41] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.

Anexos

I Anexo A: Información que almacena un chunk

Cada *chunk* almacena toda la información relacionada manteniendo la siguiente estructura:

- Versión del formato
- Nivel:
  - Localización en el mundo (x, z)
  - Entidades presentes en el *chunk*
  - *Tile entities* presentes en el *chunk*
  - Mapa de biomas
  - Mapa de altura
  - Lista de secciones:
    - Bloques (Blocks/Data)
    - Luz (Block/Sky)
  - Estado de población (Terrain/Light)
  - Tiempo habitado
  - Tiempo de la última actualización

II Anexo B: Orden de iteración del HashSet de JDK 8

El `java.util.HashSet<T>` es en realidad un wrapper de `java.util.HashMap<T, Object>`. En este caso se recorre usando `HashSet.iterator` que devuelve el iterador del `HashMap$KeySet` subyacente implementado en la clase interna `HashMap$KeyIterator`.

```
1  public class HashSet<E>
2      extends AbstractSet<E>
3      implements Set<E>, Cloneable, java.io.Serializable
4  {
5      private transient HashMap<E,Object> map;
6      ...
7
8      /**
9       * Returns an iterator over the elements in this set.  The elements
10      * are returned in no particular order.
11      *
12      * @return an Iterator over the elements in this set
13      * @see ConcurrentModificationException
14      */
15     public Iterator<E> iterator() {
16         return map.keySet().iterator();
17     }
18
19     ...
20 }
```

Código 15: `java.util.HashSet.iterator` afirma que no garantiza un orden de iteración determinado pero, mientras no cambie nuestro entorno, es fijo.

```
1 final class KeySet extends AbstractSet<K> {
2     ...
3     public final Iterator<K> iterator()     { return new KeyIterator(); }
4     ...
5 }
```

Finalmente, `KeyIterator` itera la estructura que resulta ser una tabla hash de direccionamiento cerrado que contiene `HashMap$Node<K, V>`, es decir, que resuelve las colisiones usando listas enlazadas con los elementos con igual hash. Cuando el iterador encuentra un elemento, agota la lista enlazada antes de continuar recorriendo la tabla. Así que, en resumen, se iteran los hashes de menor a mayor y las colisiones en orden de inserción.

```
1  public class HashMap<K,V>
2      extends AbstractMap<K,V>
3      implements Map<K,V>, Cloneable, Serializable {
4
5      transient Node<K,V>[] table;
6
7      ...
8
9      abstract class HashIterator {
10         Node<K,V> next;        // next entry to return
11         Node<K,V> current;     // current entry
12         int index;             // current slot
13
14         final Node<K,V> nextNode() {
15             if (this.next == null)
16                 throw new NoSuchElementException();
17
18             this.current = this.next;
19             this.next = this.current.next;
20
21             Node<K,V>[] table = HashMap.this.table;
22             if (this.next == null && table != null) {
23                 while(this.index < table.length) {
24                   this.next = table[this.index];
25                   ++this.index;
26                   if (this.next != null) break;
27                 }
28             }
29             return this.current;
30         }
31
32         ...
33     }
34
35     final class KeyIterator extends HashIterator
36         implements Iterator<K> {
37         public final K next() { return nextNode().key; }
38     }
39
40     ...
41 }
```

Código 16: Implementación de `java.util.HashMap$KeyIterator`. El código relevante se encuentra en la clase de la que hereda, `java.util.HashMap$HashIterator`.

III Anexo C: Métodos de FastUtils relevantes

```
1  /** 2^64 * phi, phi = (sqrt(5) - 1)/2. **/
2  private static final long LONG_PHI = 0x9E3779B97F4A7C15L;
```

```
3
4  /** Quickly mixes the bits of a long integer.
5   *
6   * This method mixes the bits of the argument by multiplying by the golden ratio
7   * and xorshifting twice the result.
8   * It is borrowed from "https://github.com/OpenHFT/Koloboke", and it has slightly
9   * worse behaviour than murmurHash3 (in open-addressing hash tables the average
10  * number of probes is slightly larger), but it's much faster.
11  *
12  * @param x a long integer.
13  * @return a hash value obtained by mixing the bits of {@code x}.
14  */
15 public final static long mix( final long x ) {
16     long h = x * LONG_PHI;
17     h ^= h >>> 32;
18     return h ^ (h >>> 16);
19 }
```

Código 17: `it.unimi.dsi.fastutil.HashCommon.mix`

IV Anexo D: Código desensamblado de BitArray.setAt

Extraído arrancando el juego con los siguientes argumentos de Java:
`-XX:+UnlockDiagnosticVMOptions -XX:CompileCommand=print,net.minecraft.util.BitArray::setAt`

```
1  ; public void setAt(int index, int value)
2  ; this:     rdx:rdx   = 'net/minecraft/util/BitArray'
3  ; parm0:    r8        = int index
4  ; parm1:    r9        = int value
5
6    mov     r14d, r9d                ; r14 = value
7    mov     r13, rdx                 ; r13 = this
8
9  ; Validate.inclusiveBetween(0, this.arraySize - 1, index);
10   mov     r11d, dword ptr [rdx+1ch] ; r11 = this.arraySize
11   movsxd  r9, r8d                  ; r9 = (long) index
12   dec     r11d                     ; r11 = this.arraySize - 1
13   movsxd  r10, r11d                ; r10 = (long) (this.arraySize - 1)
14   test    r9, r9                   ; if (index < 0)
15   jnl     argexception             ;    throw IllegalArgumentException
16   cmp     r9, r10                  ; if (index > this.arraySize - 1)
17   jnl     argexception             ;    throw IllegalArgumentException
18
19 ; Validate.inclusiveBetween(0, this.maxEntryValue, value);
20   mov     r9, qword ptr [rdx+10h]  ; r9 = this.maxEntryValue
21   movsxd  rbx, r14d                ; rbx = value
22   test    rbx, rbx                 ; if (value < 0)
23   jnl     argexception             ;    throw IllegalArgumentException
24   cmp     rbx, r9                  ; if (value > this.maxEntryValue)
25   jnl     argexception             ;    throw IllegalArgumentException
26
27 ; int absolute_bit = index * this.bitsPerEntry;
28   mov     edi, dword ptr [rdx+18h] ; rdi = this.bitsPerEntry
29   mov     r10d, r8d                ; r10 = index
30   imul    r10d, edi                ; r10 *= this.bitsPerEntry
31
32 ; boundary check
```

```
33    test    r10d, r10d                ; if (absolute_bit < 0)
34    jnl     oobexception              ;    throw ArrayIndexOutOfBoundsException
35
36  ; int relative_bit = absolute_bit % 64;
37  ; fast signed modulo
38    mov     ecx, r10d                 ; rcx = absolute_bit
39    neg     ecx                       ; rcx = -rcx
40    and     ecx, 3fh                  ; rcx &= 0x3F
41    neg     ecx                       ; rcx = -rcx
42
43    mov     rax, qword ptr [rdx+20h]  ; rax = this.longArray
44
45  ; int start_long = absolute_bit / 64;
46    mov     esi, r10d                 ; rsi = absolute_bit
47  ; fast signed div 64
48    sar     esi, 1fh                  ; rsi >>= 0x1F
49    shr     esi, 1ah                  ; rsi >>>= 0x1A
50    add     esi, r10d                 ; rsi += absolute_bit
51    sar     esi, 6h                   ; rsi >>= 0x6
52
53    mov     r10d, dword ptr [rax+10h] ; r10 = this.longArray.length
54
55  ; int end_long = ((index + 1) * this.bitsPerEntry - 1) / 64;
56    inc     r8d                       ; r8 = (index + 1)
57    imul    r8d, edi                  ; r8 = (index + 1) * this.bitsPerEntry
58    mov     r11d, r8d                 ; r11 = (index + 1) * this.bitsPerEntry
59    dec     r11d                      ; r11 = (index + 1) * this.bitsPerEntry - 1
60  ; fast signed div 64
61    sar     r11d, 1fh                 ; r11 >>= 0x1F
62    shr     r11d, 1ah                 ; r11 >>>= 0x1A
63    add     r11d, r8d                 ; r11 += (index + 1) * this.bitsPerEntry
64    dec     r11d                      ; r11 -= 1
65    sar     r11d, 6h                  ; r11 >> 0x6
66
67  ; boundary check
68    cmp     esi, r10d                 ; if (start_long >= this.longArray.length)
69    jnl     oobexception              ;    throw ArrayIndexOutOfBoundsException
70
71  ; this.longArray[start_long] =
72  ;      this.longArray[start_long] & ~(this.maxEntryValue << relative_bit) |
73  ;      ((long)value & this.maxEntryValue) << relative_bit;
74    mov     r8, r9                    ; r8 = this.maxEntryValue
75    shl     r8, cl                    ; r8 = this.maxEntryValue << relative_bit
76    andn    r8, r8, qword ptr [rax + rsi * 8 + 18h] ; r8 = this.longArray[start_long] & ~r8
77    and     rbx, r9                   ; rbx = value & this.maxEntryValue
78    mov     r9, rbx                   ; r9 = rbx
79    shl     r9, cl                    ; r9 = (value & this.maxEntryValue) << relative_bit
80    or      r8, r9                    ; r8 = r8 | r9
81    mov     qword ptr [rax + rsi * 8 + 18h], r8     ; this.longArray[start_long] = r8
82
83    cmp     esi, r11d                 ;if (start_long != end_long)
84    jne     second_long               ;  jump to second_long
85  return:
86    ; omitted internal jvm code
87    ret
88
89  second_long:
90  ; int cut_point = 64 - relative_bit;
```

```
91    mov     r8d, 40h                ; r8 = 64
92    sub     r8d, ecx                ; r8 = 64 - relative_bit
93
94  ; int bits_left = this.bitsPerEntry - cut_point;
95    sub     edi, r8d                ; rdi = this.bitsPerEntry - cut_point
96
97  ; boundary check
98    cmp     r11d, r10d              ; if (end_long >= this.longArray.length)
99    jnl     oobexception            ;    throw ArrayIndexOutOfBoundsException
100
101 ; this.longArray[end_long] =
102 ;   this.longArray[end_long] >>> bits_left << bits_left |
103 ;   ((long)value & this.maxEntryValue) >> cut_point;
104   mov     r9, qword ptr [rax + r11 * 8 + 18h]     ; r9 = this.longArray[end_long]
105   mov     ecx, r8d                ; rcx = cut_point
106   sar     rbx, cl                 ; rbx = (value & this.maxEntryValue) >> cut_point
107   mov     ecx, edi                ; rcx = bits_left
108   shr     r9, cl                  ; r9 >>>= bits_left
109   shl     r9, cl                  ; r9 <<= bits_left
110   or      r9, rbx                 ; r9 |= rbx
111   mov     qword ptr [rax + r11 * 8 + 18h], r9     ; this.longArray[end_long] = r9
112   jmp     return
```