Considerando las opciones para una implementación modular, se decidió empezar creando una estructura que el programa nativo intreprete. Esto es, un simulador basado en un archivo propietario. Las expectativas con esta aproximación era lograr un rendimiento mayor al de un lenguaje de scripting sacrificando algo de flexibilidad. Sin embargo se encontraron varios problemas con esta técnica, por lo que el proyecto se replanteó para poder lograr una implementación que permita definir nuevas partículas con facilidad y sea fácil de distribuir. Esto sucedió dos veces, en cada iteración del simulador ciertas bases se consolidaban de manera que la siguiente implementación no fuera de cero.

A continuación se detalla cada implementación, profundizando en sus rasgos particulares.

Simulador en C++

El primer simulador fue desarrollado en C++ usando OpenGL y GLFW. La base fundamental de este sistema fue la misma para los siguientes. Se crea un array de partículas, siendo las partículas un struct con la menor cantidad de datos posibles: el tipo, la temperatura, la granularidad, clock y life_time. Estos 5 valores ocupan en total 105 bits en memoria, algo más de 13 bytes. El tipo de la partícula es un número sin signo para identificarla, representa si la partícula es arena, ácido o cualquier otro elemento. La temperatura es un número con signo que literalmente representa su temperatura, ya que al igual que Noita, este sistema se planteó para soportar interacciones químicas. El siguiente elemento, la granularidad, es otro número, esta vez de 1 byte que se inicia de forma aleatoria y modifica ligeramente el color de la partícula. Life_time como su nombre indica es una variable que almacena la duración de la partícula en el sistema en ticks. No todas las particulas usan este valor. Finalmente, clock es un valor interno que alterna cuando la partícula es procesada y permite no procesarla más de una vez en un mismo tick.

Este sistema procesa de arriba abajo y de izquierda a derecha la matriz de particulas, por lo que si una particula se mueve hacia abajo, volverá a ser procesada en las siguiente iteraciones. Alternativamente es posible tener un array separado de booleanos y ponerlo a 0 al final de cada iteración con memset, sin embargo esto resultó ser más lento que la alternativa de usar clock. En esta, el sistema tiene también un clock que alterna cada frame, solo las partícula cuyo clock coincide con el del sistema son procesadas.

Cada partícula tiene un color asociado, en función del identificdor de la particula, se escribe su color en formato RGBA8 en buffer, este se envía cada frame a la GPU para ser renderizado. Esto es común a todas las implementaciones con ligeras variaciones.

Debido a que esta implementación era un poco explorativa, se implementó en un solo hilo, con todo, esta estaba preparada para funcionar en multihilo.

Uso de un formato propieatario

Esta es la base del sistema, pero no se ha hablado que se hace en el procesado de actualizar las partículas. Durante el desarrollo identificamos un patrón común en el movimiento de las partículas: Todas tienen patrones de movimientos similares, muchas de ellas pueden o no atravesar otras. En base a esta información, se abstrajo un modelo sobre el que poder trabajar. Este modelo se materializó en un struct ParticleDefinition.

```C++
struct ParticleDefinition
{
  std::string text_id;
  ParticleProject::colour_t particle_color;
  int16_t random_granularity;
  std::vector<ParticleProject::Vector2D> movement_passes;
  Properties properties;
```

```
8   std::vector<Interaction> interactions;
9 };
```

Este struct se guarda en un array en la misma posición que el tipo de particula que representa. Es decir, si una partícula tiene tipo 3, este struct está guardado en la posición 3 del array, de forma que se pueda indexar directamente con el tipo de la partícula. Por lo tanto, estos datos son inmutables y globales a todas las partículas de ese tipo.

`movement_passes` es un array de un Vector 2D que define el movimiento base de la partícula. La mayoría de partículas van hacia abajo si están libres, luego abajo izquierda y por último abajo derecha. Esto se traduce en: (0, 1), (-1, 1), (1, 1). Además de esto, existe un array de propiedades, entre las cuales se encuentra la densidad. Este valor se usa en el sistema y permite que partícula como la arena se hundan en el agua. Finalmente, hay un array de interacciones. Este es el que guarda la información sobre como se relacionan las partículas, el agua se evapora con la lava, la lava quema la madera, el ácido corroe la mayoría de partículas, etc.

Sin embargo, en este punto se encuentran varios problemas. Algunos de ellos son solucionables, por ejemplo, no es posible tener un movimiento aleatorio, siempre es determinista. Algunas partículas dependen de poder moverse en una dirección aleatoria en cada iteración. El siguiente problema es el que llevó a la decisión de considerar otra implementación. Esto es, las interacciones. Existen muchas posibles interacciones, agua con agua, agua con fuego, agua con planta, arena con lava, etc. Modelar una estructura de datos que permita definir interacciones de forma escalable resultó ser complejo y muy limitante. Debido a esto, se decidió implementar las interacciones con Lua.

Scripting con Lua

Se decidió mantener el código actual y solo delegar las interacciones a Lua, pues se tenía evidencia de como delegar trabajo a Lua no era adecuado para maximizar rendimiento. Se usó LuaBridge para exportar nuestros tipos de C++ a Lua. Ahora, cada ParticleDefinition tiene asociado un script de Lua cargado en memoria que recibe un objeto Api y la posición actual de la particula. Este objeto tiene un método que permite obtener el tipo de una partícula mediante su nombre. Esto permite que el script pueda interactuar con otras partículas sin saber que posición tienen en el array o si existen.

Dicha técnica otorgó mucha flexibilidad y el sistema seguía siendo sencillo de distribuir. No obstante, el rendimiento fue bastante inferior a lo esperado. Esto no era debido a que Lua fuera lento, el problema de rendimiento venía de la comunicación de Lua y C++. Más tarde se descubrió la librería Sol, que tiene menos overhead que luabridge. Aún con esto, el rendimiento seguía siendo subpar. Cabe mencionar que cuando se menciona Lua, se refiere a LuaJIT.

Nuestras pruebas nos mostraron que LuaJIT, aún siendo mucho más lento que C++, tenía un rendimiento decente. Debido a esto se optó por probar una implementación puramente en Lua.

Simulador en Lua con LÖVE

Para poder gestionar la entrada de usuario y el renderizado de forma sencilla, se usó el framework LÖVE. Esta decisión resultó ser muy beneficiosa. Además, LÖVE usa exclusivamente LuaJit.

Implementación base

La implementación en LÖVE es muy similar a la de C++, pero con muchas mejoras. Se decidió que el update de la partícula sea de implementación libre, es decir, ya no existe una estructura ParticleDefinition de la misma forma que en la implementación anterior. La motivación tras este cambio es, que si las interacciones son flexibles, no tiene sentido restringir el movimiento a una estructura de datos. Esta vez, la estructura solo contiene 3 valores, el nombre de la partícula como string, el color y una función que es toda su interacción. Por otro lado, para definir la partícula, se

usa la interfaz de funciones foráneas, FFI de ahora en adelante. Esta permite definir verdaderos structs en C, que son más rápidos de acceder y consumen menos memoria que las tablas de Lua.

```lua
-- Particle.lua

ffi.cdef [[
typedef struct { uint8_t type; bool clock; } Particle;
]]

local Particle = ffi.metatype("Particle", {})
```

Para poder aprovechar la caché, se decidió solo tener el tipo y el clock como campos de la partícula. El tipo es un uint8_t, esto permit tener hasta 256 tipos de partículas, lo cual es suficiente además de fácilmente ampliable de ser necesario.

Estos datos están guardados en una tabla global. El update de la partícula consiste en indexar esta tabla, acceder a la función de interación y llamarla pasándole un objeto API. En la implementación anterior, el API era la clase que contenía la simulación y sus métodos. En este caso se mejoró, el API es una vista de la simulación, contiene todas las particulas compartidas mediante un puntero pero solo procesa un rango, además, tiene estado interno. El API contiene la posición actual que se está procesando, por lo que las funciones que la consumen tratan con coordenadas locales. Mover una partícula hacia abajo consiste en algo similar a llamar a api.move(0, −1). Además, como cada partícula puede ser procesada de forma libre, es posible generar números aleatorios para el movimiento.

El punto más importante de esta simulación es su facilidad de añadir particulas. Como estas son definidas en Lua, es posible cargar un fichero de Lua que contenga la información de una nueva partícula. Se implementó la función de poder arrastrar archivos a la ventana. En caso de ser un archivo de Lua, este es cargado y añadido a la tabla global de partículas. Si la partícula ya existe, se sobreescribe. Este sistema permitió iterar de forma rápida el funcionamiento de las partículas.

A pesar de que todo iba bien, esta implementacion resultó ser más lenta que la realizada en C++ puro, pero más rápida que la realizada en C++ con Lua. La simulación aún tenía ciertas asperezas, entre otras, como siempre se procesan las partículas en el mismo orden, existe cierto sesgo en la simulación. Ambos problemas fueron resueltos mediante la implementación de multithreading. Se espera lograr al menos poder simular 300*300 particulas a 60fps en dispositivos de escritorio de gama baja.

Multithreading en Lua

Si bien Lua es un lenguaje muy sencillo y ligero, tiene ciertas carencias, una de ellas es el multithreading. En la versión de C++ no puedo implementarse debido a que la máquina virtual de Lua no puede ser compartida de forma segura entre hilos, de hacerlo, colapsaría. La alternativa a esto es instanciar una máquina virtual de Lua para cada hilo, esto es exacamente lo que love.threads hace. LÖVE permite crear hilos en Lua, pero además de esto, permite compartir datos entre hilos mediante `love.bytedata`. Además de esto, love proveee canales de comunicación entre hilos, que permiten enviar mensajes de un hilo a otro y sincronizarlos. Esto permitió enviar trabajo a los hilos bajo demanda.

Pese a que LÖVE facilita crear y comunicar hilos y la implementación sea sencilla, existen problemas notorios. Antes de poder explicarlos, es necesario mostrar como se implementó el multihilo en esta simulación. La forma más sencilla de actualizar la simulación, es que cada hilo actualicé un trocito o chunk en un patrón de ajedrez.

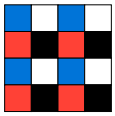

Figure 1: Patrón de ajedrez

Primero se procesan los chunks azules, luego los negros, luego los blancos y por último los rojos. Es decir, se alternan filas y columnas. Esto permite que las partículas no sobreescriban otras que estén siendo procesadas. Por lo tanto, hay 4 pases de actualización. La divisón de la simulación en grid se realiza automáticamente el base al número de cores del sistema y al tamaño de la simulación. Un chunk debe ser como mínimo de 16 * 16 pixeles. Esto permite que una partícula pueda interactuar con partículas que no estén inmediatamente cerca sin que acceda a al chunk que está siendo procesado por otro hilo.

Este sistema permite procesar chunks de particula de manera simultánea, aprovechando los recursos de la CPU y mejorando el rendimiento de la simulación. Sin embargo, existe un grave problema que está ligado al orden de actualización de simulación y el mencionado sesgo que esto conlleva.

Para ilustrar el problema, se mostrarán 3 ticks de una simulacón pequeña que solo se divide en 4 chunks. En este ejemplo no hay simulación en paralelo al ser de una escala tan pequeña, pero se puede ver como el orden de actualización afecta a la simulación. El orden de procesado global y dentro de cada chunk es, de derecha a izquierda y abajo a arriba. Para este ejemplo primero se procesa el chunk superior derecho, luego inferior izquierdo, luego superior izquierdo y por último inferior derecho. Este orden es distinto al dado anteriormente, pero este problema sucede independientemente del orden en función del movimiento de las partículas. La partícula mostrada solo tiene un comportamiento, moverse hacia abajo si no hay otra partícula.

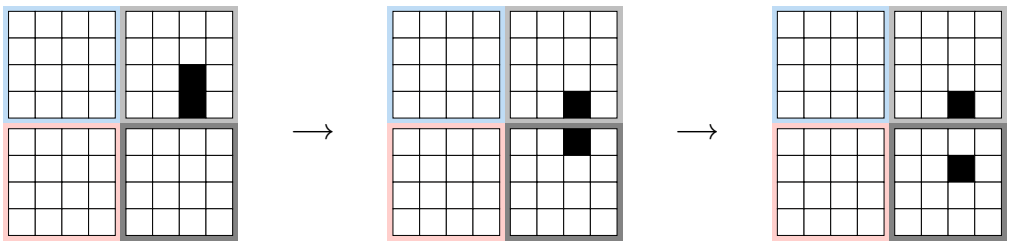

Figure 2: Problema de multithreading

El tercer tick ya deja ver el problema. Si la simulación fuera single thread, el estado de la simulación sería el siguiente:

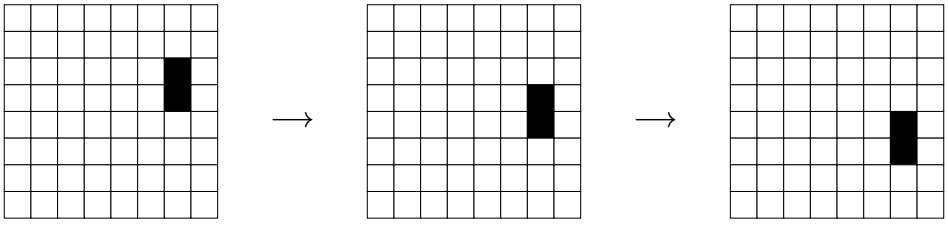

Figure 3: Resultado esperado

El problema radica en el orden de actualización tanto de las partículas como de los chunks. Como se procesa primero el bloque superior, la partícula detecta que hay una debajo y no se mueve. En un

solo hilo, como la actualización es de abajo hacia arriba, esto no ocurre. Este problema está presente siempre independientemente del orden de actualización elegido. Este problema sucede incluso si se cambia el orden de actualización de las partículas en cada iteración.

Este problema no tiene solución, no es posible obtener el resultado esperado mostrado en la figura superior. Lo que sí es posible es justo lo contrario. Si el efecto mostrado en el problema del multithreading sucede en cualquier frame y no solo en la frontra de los chunks, entonces visualmente la simulación sería consistente y no se podría apreciar el orden de procesado. Para realizar esto hay que cambiar tanto el orden de simulación como el de procesado de los chunks. Solo es necesario tener dos lotes: El primero comienza procesando el chunk superior izquierdo, luego el inferior derecho, luego el superior derecho y por último el inferior izquierdo, las partículas se simulan de izquierda a derecha y de arriba a abajo. El segundo lote es todo lo contrario, se comienza por el chunk inferior izquierdo, luego el superior derecho… Las partículas se procesan de derecha a izquierda y de abajo a arriba.

Esta solución minimiza mucho el efecto visual provocado por procesar la simulación en chunks, sin embargo, aún es posible ver artefactos notables. Esto es debido a que cuando una partícula se mueve a un chunk no procesado, desplaza o mueve a la partícula que estaba en su lugar y bloquea la ejecución en dicha coordenada. Para solucionar esto la solución fue implementar un doble buffer de partículas, como en el Juego de la Vida. En este juego, se lee el primer buffer y se escribe el resultado en el segundo, de esta forma, que un pixel cambie mientras se procesa la simulación no afecta a los demás. Esto es posible hacerlo porque en dicho autómata celular cada partícula solo modifica la coordenada en la que está. Sin embargo, en una simulación más flexible y complejo como es esta, una partícula puede moverse a otra coordenada o incluso alterar a sus vecinas (fuego quemando planta).

El funcionamiento del doble buffer es el siguiente. Existe un buffer de partículas de solo lectura y otro de escritura. Al terminar un tick de la simulación, el buffer de escritura se copia al de lectura y se procesa el siguiente frame. Como el buffer de escritura es inmutable, al modificar una partícula, se modifica el clock de la coordenada correspondiente en el buffer de escritura. Este también se usa como atributo de lectura. Para ver si una coordenada está vacía, se accede al buffer de escritura y además se verifica que el clock de la matriz de escritura no esté a true. En caso de que lo esté, es que otra partícula ya modificó esa posición por lo que los datos del buffer de lectura no son fiables. Esto es un comportamiento excepcional para comprobar sin un hueco está vacío, para comprobar el tipo de una particula en una posición dada, simplemente se accede al buffer de lectura. Es posible que haya sido escrita por otra partícula, pero desde el punto de vista de la simulación solo es relevante que era esa partícula al inicio, independiente de que haya sido modificada por otra porque justo el orden de actualización de la iteración actual provocara que la otra partícula fuera simulada antes.

Con esto el sistema funciona satisfactoriamente, es suficientemente rápido y fácil de extender mediante mods que son arrastrar y soltar. Además, es fácil de distribuir, solo es necesario un ejecutable de LÖVE y los archivos de Lua. El mayor problema de este sistema es la escalabilidad, si las partículas son muy complejas o hacen muchas operaciones, es posible que algunos dispositivos tengan problemas incluso con simulaciones pequeñas. Esto es visible al tener partículas como la de fuego, que itera todos sus vecinos buscando una planta que quemar. Esto también se puede observar si se implementa el juego de la vida sobreescribiendo la particula de vacío por una “célula muerta” y añadiendo una “célula viva”, pues en este caso, independiente del estado de la simulación todas las partículas comprueban sus vecinos. Debido a esto, se decidió implementar la última alternativa a un sistema modular: Un sistema que funcione en base a librerías dinámicas. Para este sistema se decidió usar Rust y Macroquad.

Quizás entre estos párrafos pueda poner imágenes de la simulación en distintos puntos para que no sea solo texto, no estoy seguro

Hay explico las soluciones directamente sin profundizar mucho en detalles técnicos porque sino este capítulo ocuparía el triple que toda la memoria, además de que en este caso particular, explicar como llegué a este conclusión es complejo, pero puedo hacerlo. Hay problemas que no he mencionado que no fueron triviales de detectar, como el extraño funcionamiento de punteros al usar la FFI de LuaJIT, pero como eso es algo más de programación general que específico de la simulación no lo mencioné. No sé si este capítulo es demasiado largo, lo he hecho corto, podría hacerlo muchísimo más largo. No me gusta mucho este capítulo porque parece un poco un devlog pero es que no sé si no como hacerlo :(Tampoco he mencionado como es posible interactuar con una partícula de Tipo X sin saber su id mediante una tabla global ParticleType que se crea automáticamente y mapea nombre de partícula a stirng permitiendo hacerte un ParticleType.ParticleName.

Simulador en Rust con Macroquad

Voy a resumir muchísimo aquí, porque aunque esta simulación me llevó 3 días contado (y luego pulí detalles en los siguientes), la cantidad de información que contiene este proycto es inmensa, fueron 3 días muy intensos en los que no dormí nada y aprendí muchísimo sobre Rust y Wasm. A pesar de que he mencionado Rust para el TFG en otras ocasiones, en verdad casi no tengo experiencia con este lenguaje. Afortunadamente el libro de Rust es una bendición y nuestra formación y conocimientos previos hicieron muy fácil adaptarse a este lenguaje que aunque parezca otro lenguaje más no tiene nada que ver con C, C++, Java, Python, Lua o cualquier otro lenguaje orientado objetos o no orientado a objetos que usa objetos de todas formas porque los programadores de hoy en día parecen no conocer otra cosa (y yo no es que sea diferente, si no uso OOP me muero, de programación funcional, mónadas y esos constructos sé lo justito)

Después de haber implementado la misma simulación tres veces diferentes (y varias más que no resultaron en nada), la última versión fue en Rust por varias razones. La primera es la gestión de dependencias. Crear un proyecto en Rust es más rápido que en C++ aún usando Conan o vcpkg. La segunda es WebAssembly. Rust fue uno de los lenguajes que más pronto adoptaron WebAssembly y cuenta con muchas facilidades para su desarrollo: `wasm-bindgen`, `wasm-pack`, `trunk`... Entre otros. Por otro lado, Macroquad es un framework de Rust inspirado por raylib. Es multiplataforma y compila de forma nativa a WebAssembly, por lo que el mismo código puede ser usado para la versiòn web y la nativa. Como desventaja, el game loop viene hecho y no permite cambiar la tasa de refresco, la justificación del desarrollador es que en WebAssembly la tasa de refresco depende de la función de Javascript `requestAnimationFrame` usada para implementar el game loop. A pesar de este inconveniente, Macroquad es muy sencillo de usar y tiene una documentación muy completa.

La migración de LÖVE a Macroquad fue bastante sencilla, se reutilizó la mayoría de la arquitectura con adaptaciones a Rust a excepción del multithreading, ya que este es más complejo de implementar en Rust y no es compatible con WebAssembly. El sistema de plugins funciona igual que en Lua, sustituyendo los archivos Lua por librería dinámicas. En función de la plataforma el programa aceptará DLL, so, o dylib. Además de esto, se amplió la funcionalidad de los plugins, ya que en Rust son un struct que implementan el trait plugin. Esto permite que cada plugin tenga su propio estado. Las partículas, por su parte tienen los mismos campos que en Lua, id y clock, pero además se añadieron otros dos: light y extra. Light es la opacidad de la particula, útil para que algunas particulas tengan ligeras variaciones y no se vea plano. Extra son 8 bits para guardar información. En total, cada partícula ocupa 4 bytes. Clock es una variable de 1 byte, aunque solo se usa 1 bit y los otros 7 quedan inutilizados.

Tengo que reestructurar esto mejor porque tengo que hablar de que en Rust uso el ABI de Rust con las consecuencias que ello conlleva

Como gestionar proyectos en Rust es relativamente sencillo con cargo, el proyecto se organizó en 3 crates: plugins, app-core y app. Plugins contiene plugins por defecto para inicializar la aplicación con algo más que una particula vacía, app-core contiene toda la lógica de la simulación y el trait de plugin, app es un crate binario que usa app-core para crear la simulación y renderizarla. La simulación tiene un array de bytes que guarda los colores, pero no tienen imagen ni textura, esta se crea en el crate de la aplicación, de esta forma la lógica de la simulación es totalmente independiente de la representación visual y de macroquad.

El rendimiento de este proyecto es muy superior al de Lua aún usando un solo hilo. Como desventaja, al distribución es más compleja, aunque es posible realizar compilación cruzada mediante acciones de GitHub, ahora hay que distribuir los plugins en 3 formatos distintos. Si se quisiera hacer una especia de tienda online de plugins para que otros usuarios compartan los suyos, sería molesto obligarles a compilar para distintas plataformas. Esto es fácilamente solucionable, se les puede proveer una imagen Docker que realiza la compilación, se puede crear un editor online que internamente envíe el código a un servidor y lo compile… Existen posibilidades de paliar este problema y facilitar a los usuarios o a nosotros como desarrolladores la creación y distribución de plugins. Sin embargo, existe una opción mejor, la opción más equilibrada de todas y al mismo tiempo, la más compleja.

WebAssembly es un concepto muy interante para los desarrolladores, es un destino de compilación que alcanza un rendimiento muy decente y tiene una distribución y alcance enormes al poder compartir tu apliación mediante un enlace. No obstante, no todo es tan bonito, aunque el rendimiento en WebAssembly sea muy superior al de LuaJIT nativo, no soporta multithread de forma sencilla, ya que usar los web worker desde Rust es un tema que sigue en desarollo y no es estable (tengo que poner referencia aquí, lo leí en algún lado). Además de esto, la carga de plugins es compleja. En la versión de Lua era arrastrar y soltar, en la versión nativa de Rust con Macroquad esto también es posible aunque actualmente el ejecutable carga todos los plugins que están en la raiz donde se encuentra. En WebAssembly esto es más complejo, un módulo WebAssembly puede haber sido compilado desde C++, o desde C#… Es el mismo problema que las DLLs pero peor, en este caso aunque existe un ABI estable, el de C, no es tan sencillo como usarlo. Esto depende de si el módulo fue compilado con emscripten u otra herramienta, debido a que puede haber hecho un name mangling distinto, puede haber organizado el stack de distinta forma, puede haber enlazado los símbolos de forma distinta… Debido a esto, la comunicación entre módulos es compleja aún cuando son compilados de la misma forma, con el mismo lenguaje y mismo toolchain. WebAssembly solo permite pasar como parámetro números y punteros. Las funciones en WebAssembly solo pueden recibir un parámetro, el compilador crea funciones para poder solventar esto. Actualmente no fue posible pasar un Plugin de un módulo WebAssembly al principal. Además de esto, no se está seguro de si es posible por las razones expuestas, y en caso de serlo, no se sabe si tendría penalización del rendimiento, ya que WebAssembly, al acceder a un espacio de memoria que no le pertenece es posible que tenga que acceder a través del host, lo cual sería lento.

De ser posible, la versión en Rust exportada a WebAssembly sería la más prometedora. Sería eficiente, flexible y fácil de distribuir. Además, de ser posible, existe la posibilidad de recrear la estructura de plugins en AssemblyScript, ya que de esta forma podría crearse un editor visual por bloques y compilar en runtime para poder actualizar los plugins y tener un sistema realmente flexible que permita prototipar de forma rápida y sencilla.

Wasm es todo un mundo, no he encontrado aún información sobre esto, por eso no estoy seguro, lo que he leído sobre wasm no es nada esclarecedor.

Estas implementaciones, en mayor o menor medidas están limitadas por la CPU, al ser tantas partículas que procesar se trata de simplificar una particula para que ocupe menos espacio y por tanto más parte del array pueda ser almacenado en la RAM. Además se trata de usar multithreading para aumentar el rendimiento, pero las CPUs actuales no tienen una cantidad de núcleos inmensas, y aún de tenerlas procesar las partículas sin artefactos visuales sería complejo por lo expuesto en el apartado de Lua. A pesar de esto, existen autómatas celulares que se computan en la GPU, por lo que existe la posibilidad de lograr una simulación superior a la de GPU al menos en lo que a cantidad de partículas simuladas se refiere. Para investigar esto se realizó una implementación en la GPU con Vulkan.