TFG Noita

Nicolás Rosa Caballero
Jonathan Andrade Gordillo

February 26, 2024

Índice

No sé si esta es la mejor forma de crear una tabla de figuras pero es lo que encontré por ahora. Es posible crear una tabla usando una query (figure.where) y dándole el formato que nosotros queramos.

Índice de Figuras

1 Introduction

2 Autómatas Celulares y simuladores de arena

Fuente del siguiente párrafo: https://en.wikipedia.org/wiki/Cellular_automaton No consigo averiguar de donde sacó wikipedia la información, las fuentes que cita son innacessibles o bien no contienen la información que necesito.

John von Neumann, matemático húngaroestadounidense, investigaba el problema de sistemas auto replicantes en la década de 1940. Su diseño se basaba en la idea de un robot que construye a otro. Esto le resultaba complejo debido a la dificultad de tener que proveer a dicho robot de un gran conjunto de piezas que pudiera usar para replicarse. Tras la publicación de un papel científico en 1948 titulado «The general and logical theory of automata», un colega suyo, Stanislaw Ulam, matemático polaco; sugirió usar un sistema discreto para crear un modelo reduccionista de auto replicación.

Más tarde, en 1950, Ulam y von Neumann crearon un método para calcular el movimiento de los líquidos. El concepto impulsor de dicho método consistía en considerar un líquido como un grupo de unidades discretas y calcular el movimiento de ellas basándose en los comportamientos de sus vecinas. Así nació el primer autómata celular No he encontrado uno anterior ni en wikipedia ni fuera de ella. No parece haber mucha información sobre los orígenes de los autómatas celualres

Bueno, al menos el siguiente párrafo tiene una fuente más sólida, aunque parte de la información del párrafo proviene de la wiki de Gardner

Los descubrimientos de von Neumann y Ulam propiciaron que otros matemáticos e investigadoras contribuyeran al desarrollo de los automátas celulares, sin embargo, no fue hasta 1970 que se popularizaron de cara al público general. En este año, Martin Gardner, divulgador de ciencia y matemáticas estadounidense; escribió un artículo [1] en la revista Scientific American sobre un automáta celular específico: El juego de la vida de Conway.

Los autómatas celulares [2] son un modelo matemático que consiste en una matriz n-dimensional de celdas. Cada celda puede estar en un estado de un conjunto de estados finitos. Estos estados cambian en función de los estados de sus celdas vecinas cada unidad discreta de tiempo llamada generación [3]. Cada celda tiene 8 vecinos, laterales y esquinas inclusive. Al procesar un automáta celular, una celda y sus vecinos se consideran como una sola, esto es llamado "vencidad de Moore"[4]. Si solo se consideran los vecinos sin las esquinas entonces es llamado "vencidad de Neumann"[4]. Por último, para definir un autómata celular necesitamos una "regla" que aplicar a cada celda, esta transformará el sistema de celdas en el siguiente estado en cada generación.

A continuación se mostrará un ejemplo de automáta celular para comprender mejor su concepto y propiedades. Este sistema consiste en una matriz bidimensional infinita cuyas celdas pueden tener cuatro estados posibles que serán llamados `agua`, `lava`, `roca` y `vacío`. Se presupone un sistema de coordenadas (X,Y). La coordenada X representa la posición horizontal de la celda de izquiera a derecha. La coordenada Y representa la posición vertical de la celda de abajo a arriba. Para simplificar la explicación de este sistema se definirá la operación mover. Se entiende como moverse a la transformación matemática por el cual se toman dos coordenadas, `inicio` y `final`, en la que la que el estado de la celda final se sobreescribe con el de la celda inicio. La celda inicio pasa a tener como estado `vacío`. (no sé si debería escribir esto en notación matemática, tampoco estoy muy seguro de como sería dicha notación). Una celda en estado agua puede moverse hacia debajo si la celda inferior contiene estado `vacío`. Si la celda inferior contiene el

estado `lava`, la celda inferio pasará a ser estado `roca` y la celda actual pasará de estado `agua` a `vacío`. Una celda en estado `roca` no sufre alteraciones de ningún tipo. Una celda en estado `lava` se comporta similar a una celda con estado `agua`. Puede moverse hacia abajo si la celda inferior está vacía y se transforma en roca si la celda inferior es `agua`. Para representar el estado de forma visual, se asigna el color rojo a la celda en estado lava, azul al estado agua, negro al estado roca y blanco al estado vacío:

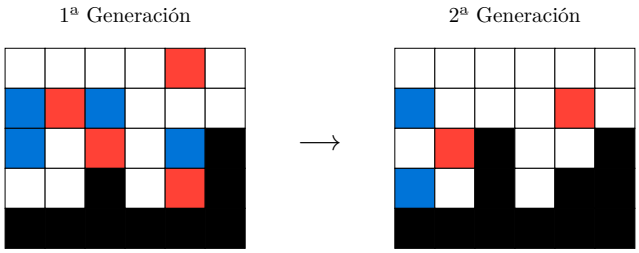

Figura 1: Ejemplo de autómata celular

Los automátas celulares han interesado a la comunidad científica y matemática desde su establecimiento por von Neumann. Debido a esto existen diversos autómatas celulares desarrolladores por distintos investigadores. A continuación se muestran una serie de autómatas celulares relevantes.

- Autómata de Contacto [4]

Un autómata de contacto puede considerarse como uno de los modelos más simples para la propagación de una enfermedad infecciosa. Imagina una cuadrícula cuadrada de celdas. Supongamos que todas las celdas son «blancas» (o «no infectadas») excepto un número finito de celdas «negras» (o «infectadas»). Para una celda dada, definimos los vecinos como las ocho celdas inmediatamente adyacentes y la celda misma. Una versión determinista de un proceso de contacto funciona en pasos discretos de tiempo de la siguiente manera: una celda negra permanece negra, una celda blanca que es vecina de una celda negra se vuelve negra. Este sistema puede denominarse un proceso de contacto determinista. Si comenzamos con una sola celda negra, entonces el número de celdas negras aumenta en 1, 9, 25, etc. La forma general sigue siendo cuadrática. Se usa en la investigación de propagación de fenómenos epidemiológicos.

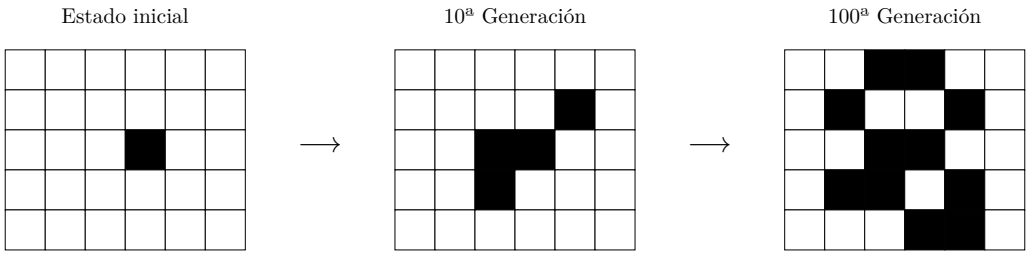

Figura 2: Ejemplo de autómata de contacto

- Autómatas de Wolfram [4]

Los Autómatas de Wolfram, propuestos por el físico y matemático Stephen Wolfram, son un conjunto de reglas para autómatas celulares unidimensionales de estados binarios. Cada regla define cómo evolucionan las células en función de su estado y el estado de sus vecinos. Cada

generación en un automáta de wolfram es una fila más qu ese añade a las anteriores. Numeradas del 0 al 255, estas reglas proporcionan un marco teórico para explorar la complejidad emergente y la computación universal en sistemas celulares simples.

A continuación se muestra la regla 30 [5] de Wolfram así como el autómata derivado de ejecutar 15 iteraciones de este:

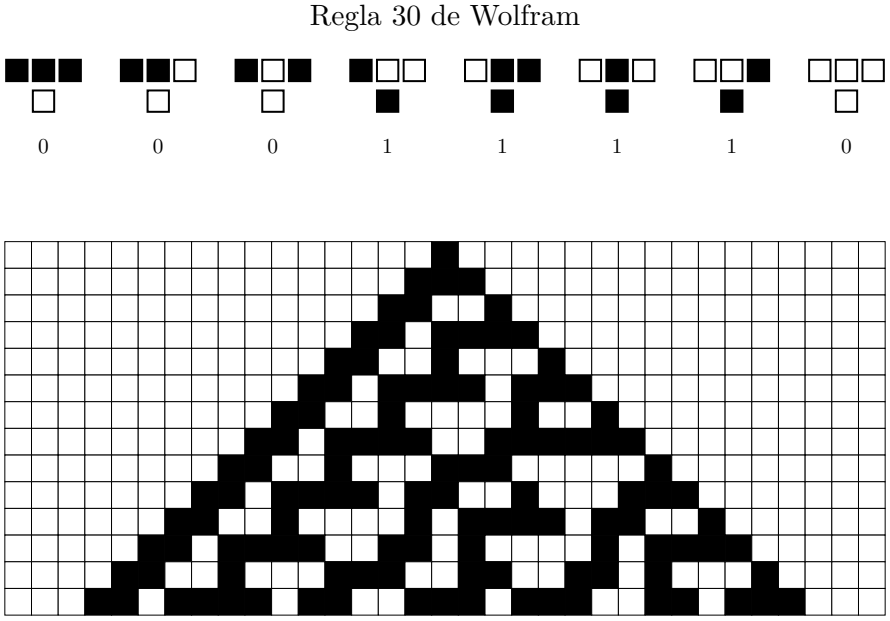

Figura 3: Ejemplo autómata de Wolfram

- Autómata de Greenberg-Hastings [4]

Los autómatas de Greenberg-Hastings son bidimensionales y están compuestos de células que pueden estar en 3 estados: reposo, excitado y refractario. La evolución de las células se basa en reglas locales que determinan la activación y desactivación de las células en función de su estado actual y el estado de sus vecinos. Debido a esteo este modelo ha sido utilizado para simular patrones de propagación de la actividad eléctrica en tejidos cardiacos. Destacando por su capacidad para modelar fenómenos de propagación y ondas, este autómata sigue reglas específicas para la activación y desactivación de células, lo que lo convierte en una herramienta valiosa en la investigación biomédica.

TODO: Añadir ejemplo de Greenberg-Hastings (no hay ninguno en el libro ni en internet, no esoty muy seguro de como crearlo pero el concepto de este autómata es intereante, ¿Podría dejarlo sin ejeplo o tendría que quitarlo por ser el único sin ejepmlo?).

- Hormiga de Langton [4]

La hormiga de Langton es un autómata bidimensional de 18 estados. Cada celda puede ser blanca o negra, además de que puede contener o no a la hormiga (solo hay una). La hormiga está orientada a hacia una de las 4 direcciones cardinales y solo se mueve una sola vez de acorde a las siguientes reglas: Si le hormiga está en una celda negra, cambia su orientación 90 grados a la derecha. Si está en una celda blanca, cambia su orientación 90 grados a la izquierda. Finalmente, cada vez que la hormiga abandona una celda, esta cambia de color. Este sistema se vuelve periódico tras algo más de mil iteraciones, creando una estructura con forma de camino

con periodo 104. A continuación se mostrará un ejemplo de las primeras iteraciones. La hormiga se representará con el color rojo si está en una celda negra y azul si está en una celda blanca. Se asume que en el estado inicial, la hormiga está orientada hacia arriba y en una celda blanca.

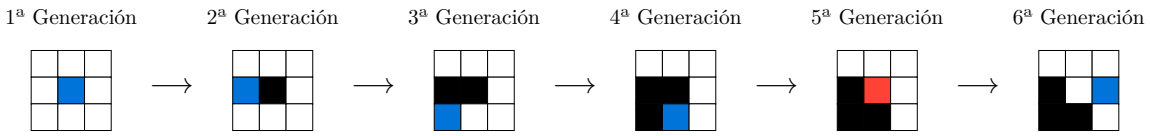

Figura 4: Ejemplo de la hormiga de Langton

- Juego de la vida

El Juego de la Vida, propuesto por el matemático John Conway, es un autómata celular bidimensional que se desarrolla en una cuadrícula infinita de células, cada una de las cuales puede estar en uno de dos estados: viva o muerta. La evolución del juego está determinada por reglas simples basadas en el número de vecinos vivos alrededor de cada célula. Este modelo ha demostrado ser notable debido a su capacidad para generar patrones complejos y estructuras dinámicas a partir de reglas simples de transición de estados. El Juego de la Vida ha sido ampliamente estudiado en el campo de la teoría de la complejidad y ha sido utilizado como un modelo para explorar la autoorganización y la emergencia de la complejidad en sistemas dinámicos.

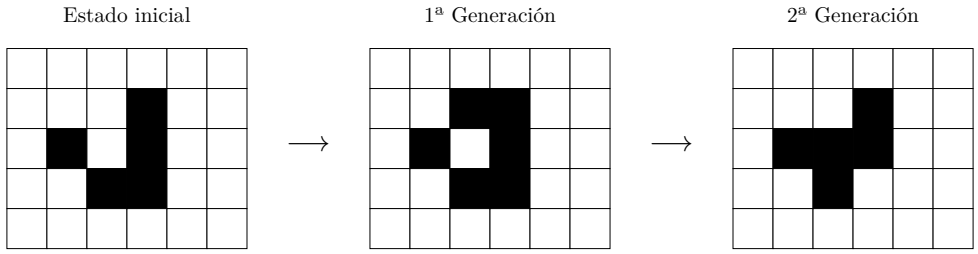

Figura 5: Ejemplo del Juego de la Vida

Con el tiempo, esos sistema llamaron la atención de un público menos científico debido a su cualidad recreacional, pues no era necesario entender los fundamentos matemáticos de estos para poder disfrutar sus interacciones. De esta forma, y con un enfoque orientado a lo lúdico, es como surgieron los simuladores de arena.

2.1 Simuladores de arena

En este capítulo, se hablará de manera resumida acerca de qué son los simuladores de partículas de manera general así como sus múltiples aplicaciones a diferentes sectores. Mas tarde, se presentan una serie de antecedentes que se han tomado de base para el desarrollo del proyecto.

2.1.1 Introducción a la simulacion de partículas

Los simuladores de partículas son un subgénero destacado tanto en el sector de los videojuegos como en el de los autómatas celulares [6], haciendose uso de ellos para herramientas y experiencias capaces de representar interacciones dinámicas presentes en el mundo real.

Los simuladores de partículas cumplen las características básicas que permiten considerarlos autómatas celulares, ya que: el «mapa» de la simulación está formado por un conjunto de celdas

dentro de un número finito de dimensiones, cada una de estas celdas se encuentra, en cada paso discreto de la simulación, en un estado concreto dentro de un número finito de estados en los que puede encontrarse y el estado de cada celda es determinado mediante interacciones locales, es decir, está determinado por condiciones relacionadas con sus celdas adyacentes.

Estos abarcan una gran diversidad de enfoques destinados a proporcionar diferentes funcionalidades y experiencias dependiendo del enfoque del proyecto. Algunos de los tipos mas reseñables son:

- Simulador de fluidos [7]: Enfoque dirigido a la replicación del comportamiento de sustancias como pueden ser el agua y diferentes clases de fluidos, lo que implica la simulación de fenómenos como flujos y olas y de propiedades físico químicas como la viscosidad y densidad del fluido.
- Simulador de efectos [8]: Categoría que apunta a la simulación detallada de efectos de partículas,como pueden ser las chispas, humo, polvo, o cualquier elemento minúsculo que contribuya a la creación y composición de elementos visuales para la generación de ambientes inmersivos.
- Simuladores de arena [9]: Conjunto de simuladores cuyo objetivo es la replicación exacta de interacciones entre elementos físicos como granos de arena u otros materiales granulares. Esto implica la modelización de interacciones entre particulas individuales, como pueden ser colisiones o desplazamientos, buscando recrear interacciones reales de terrenos.

2.1.2 Importancia y aplicación en diversos campos

La versatilidad de los simuladores de partículas ha llevado a su aplicación y adopción en una vasta gama de disciplinas mas allá de los videojuegos, debido a la facilitación que ofrecen de trabajar con fenómenos físicos de manera digital.

Algunos — aunque no todos — de los campos que han hecho uso de simuladores de particulas, así como una pequeña explicación acerca de su aplicación son:

- Física y ciencia de materiales

Los simuladores de partículas han permitido la experimentación virtual de propiedades físicas de diferentes elementos, como pueden ser la dinámica de partículas en sólidos o la simulación de materiales.

- Ingeniería y construcción

Se emplean simuladores con el objetivo de prever y comprender el funcionamiento de diferentes estructuras y materiales en el ámbito de construcción antes de su edificación, lo que permite predecir elementos básicos como la distribución de fuerzas y tensiones así como el comportamiento ante distintos fenómenos como pueden ser terremotos

- Medicina y biología

Los simuladores de partículas en el ámbito de la medicina permite modelar comportamientos biológicos así como, por ejemplo, imitar la interacción y propagación de sustancias en fluidos corporales, ayudando al desarrollo de tratamientos médicos.

2.1.3 Simuladores de arena como videojuegos

Dentro de la industria de los videojuegos, se han utilizado simuladores de partículas con diferentes fines, como pueden ser: proporcionarle libertad al jugador, mejorar la calidad visual o aportarle variabilidad al diseño y jugabilidad del propio videojuego.

Este proyecto toma como principal referencia a «Noita», un videojuego indie roguelike que utiliza la simulación de partículas como núcleo principal de su jugabilidad. En «Noita», cada píxel en pantalla representa un material y está simulado físicamente según reglas físicas y químicas específicas de ese material. Esto permite que los diferentes materiales físicos, líquidos y gaseosos se comporten de manera realista de acuerdo a sus propiedades. El jugador tiene la capacidad de provocar reacciones en este entorno, como destruirlo o hacer que interactúen entre sí.

Figura 6: Imagen gameplay de Noita

Por supuesto, Noita no es el primer videojuego que hace uso de los simuladores de partículas. A continuación, voy a enumerar algunos de los títulos, tanto videojuegos como sandbox, más notables de simuladores de los cuales hemos tomado inspiración durante el desarrollo.

- Falling Sand Game [10]

Probablemente el primer videojuego comercial de este amplio subgénero. A diferencia de Noita, este videojuego busca proporcionarle al jugador la capacidad de experimentar con diferentes partículas físicas así como fluidos y gases, ofreciendo la posibilidad de ver como interaccionan tanto en un apartado físico como químicas. Este videojuego establecería una base que luego tomarían videojuegos más adelante.

- Powder Toy [11]

Actualmente el sandbox basado en partículas más completo y complejo del mercado. Este no solo proporciona interacciones ya existentes en sus predecesores, como Falling Sand Game, sino que añade otros elementos físicos de gran complejidad como pueden ser: temperatura, presión, gravedad, fricción, conductividad, densidad, viento etc.

- Sandspiel [12]

Este proyecto utiliza la misma base de sus sucesores, proporcionando al jugador libertad de hacer interaccionar partículas a su gusto. Además, añade elementos presentes en Powder Toy como el viento, aunque la escala de este proyecto es mas limitada que la de proyectos anteriores. De Sandspiel, nace otro proyecto llamado Sandspiel Club [13], el cual utiliza como base Sandspiel, pero, en esta versión, el creador porporciona a cualquier usuario de este proyecto la capacidad de crear partículas propias mediante un sistema de scripting visual haciendo uso de la librería

blockly [14] de Google. Además, similar a otros títulos menos relevantes como Powder Game (No confundir con Powder Toy), es posible guardar el estado de la simulación y compartirla con otros usuarios. A cambio de esta funcionalidad, en Sandspiel Club no es posible hacer uso del viento, elemento sí presente en Sandspiel.

3 Programación paralela

En este capítulo, se desea introducir al lector el funcionamiento y programación de GPUs, hablando acerca de su arquitectura, es decir, cómo están fabricadas,su origen y sus usos.

3.1 Introduccion

Las Unidades de Procesamiento Gráfico, comúnmente conocidas como GPUs (por sus siglas en inglés, Graphics Processing Units), son hoy en dia un componente clave de la informática moderna presente en todos los ordenadores, destinado principalmente al procesamiento gráfico.

El origen del concepto de lo que sería una GPU hoy en día, se remonta a 1980, con un proyecto en el cual con el objetivo de generar imagenes 3D para el estudio de la estructura protéica, en la Universidad de Carolina del Norte (UNC) desarrollaron un sistema llamado Pixel Planes, el cual tuvo la innovadora aplicación de asignar un procesador por pixel mostrado, lo que significaba que muchas partes de las imágenes en la pantalla se generaban simultáneamente, mejorando enormemente la velocidad en la cual se generaban los gráficos.[15] Este trabajo continuó hasta 1997 donde se desarrolló la última iteración del proyecto.

En 1982, Nippon Electric Company introdujo el primer controlador gráfico a gran escala, el chipset gráfico NEC µPD7220, lanzado originalmente para la PC-98. Este controlador introdujo aceleración hardware para dibujar lineas, círculos y otras figuras geométricas en una pantalla de píxeles.

Hasta ese momento, se utilizaban terminales caligráficas para mostrar dibujos e imagenes mediante trazos vectoriales en lugar de píxeles individuales, ya que la mayoría de microcomputadores carecían de la potencia necesaria para hacer uso de pantallas rasterizadas, que utilizan una cuadrícula de píxeles para representar imágenes en pantalla.

Figura 7: Geforce 256, la primera tarjeta gráfica independiente

El término GPU no se popularizaría hasta el año 1999, donde NVIDIA sacó y comercializó la GeForce 256 [16] como la primera unidad de procesamiento gráfico completamente integrada que ofrecía transformaciónes de vértices, iluminación, configuración y recorte de triángulos y motores de renderizado en un único procesador de chip integrado. El chip contaba con numerosas características avanzadas, incluyendo cuatro pipelines de renderizado independientes. Esto permitía que la GPU alcanzara una tasa de relleno (cantidad de píxeles que una GPU escribe sobre la memoria de vídeo para crear el fotograma sobre esta) de 480 Megapíxeles por segundo. La salida de video era VGA. Además, el chip incluía mezcla de alfa por hardware y cumplía con los estándares de televisión de alta definición (HDTV). Lo que de verdad diferenció a la

GeForce 256 de la competencia fue la integración de iluminación que lo diferenció de modelos pasados y de la competencia, que hacían uso de la CPU para ejecutar este tipo de funciones, mejorando el rendimiento y abaratando costes para el consumidor.

3.2 Arquitectura

El propósito principal de una GPU es priorizar el procesamiento rápido de instrucciones simples mediante una estrategia de «divide y vencerás». Esto implica dividir la tarea en componentes más pequeños que pueden ser ejecutados por los numerosos núcleos presentes en una GPU. A diferencia de los núcleos de la CPU, los núcleos de la GPU son más lentos y menos versátiles, con un conjunto de instrucciones más limitado. Sin embargo, la compensación radica en el mayor número de núcleos disponibles para ejecutar instrucciones en paralelo. [17]

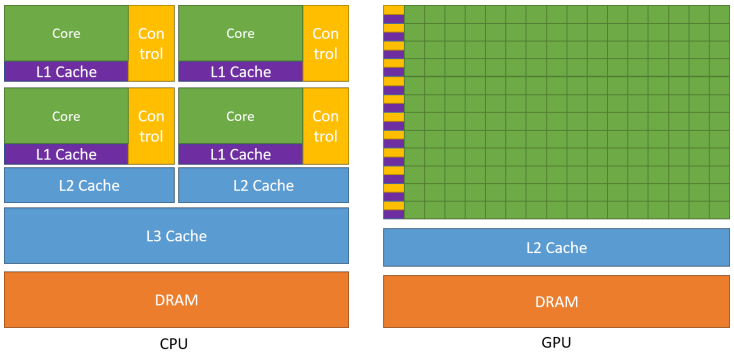

Figura 8: Comparativa arquitectura de un chip de CPU y de GPU

Se hará uso de CUDA para explicar el funcionamiento de la ejecución de programas en GPU.[18]

La GPU está optimizada para realizar tareas de manera paralela, mientras que la CPU está diseñada para ejecutar tareas de forma secuencial. Esta diferencia se refleja en la distribución del espacio dentro del chip. En una GPU, la mayor parte del espacio se dedica a tener muchos núcleos pequeños, mientras que en una CPU, la estructura está más orientada hacia una jerarquía de memoria, con múltiples niveles de caché y núcleos más grandes, potentes y capaces, pero limitados en términos de paralelización.

La arquitectura de la GPU se compone de múltiples SM (Streaming Multiprocessors), que son procesadores de propósito general con baja velocidad de reloj y una caché pequeña. La tarea principal de un SM es controlar la ejecución de múltiples bloques de hilos, liberándolos una vez que han terminado y ejecutando nuevos bloques para reemplazar los finalizados. Cada SM está formado por núcleos de ejecución, una jerarquía de memoria que incluye: caché L1, memoria compartida, caché constante y caché de texturas, un programador de tareas y un número considerable de registros.

Figura 9: Streaming Multiprocessor

Desde el punto de vista de software, la ejecución de programas se divide en kernels, grids de bloques, bloques de hilos e hilos.

Se le conoce como kernel a aquellas funciones que están destinadas a ser ejecutadas en la GPU desde la CPU o Host. Estas funciones se subdividen en un grid de una, dos o hasta 3 dimensiones de bloques de hilos mediante los que se ejecutará el kernel. El número de bloques totales que se crean viene dictado por el volumen de datos a procesar. Es totalmente necesario que estos bloques de hilos puedan ser ejcutados de manera totalmente independiente y sin dependencias entre ellos, ya que a partir de aquí el programador de tareas es el que decide que y cuando se ejecuta. Los hilos dentro del bloque pueden cooperar compartiendo datos y sincronizando su ejecución para coordinar los accesos a datos, teniendo una memoria compartida a nivel de bloque. El número de hilos dentro de un bloque esta limitado por el tamaño del SM, ya que todos los hilos necesitan residir en el mismo SM para poder compartir recursos de memoria.

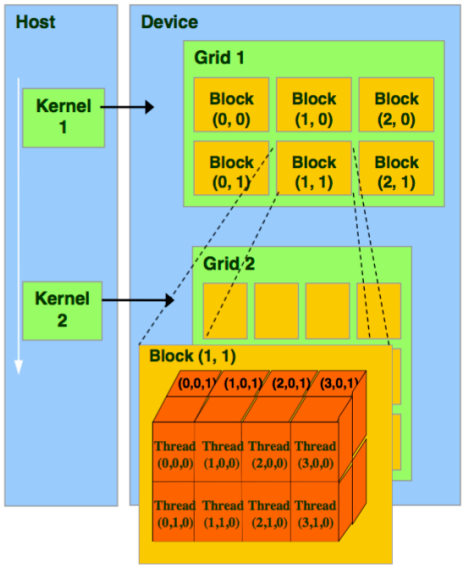

Figura 10: Jerarquía de ejecución

3.3 Usos

La función inicial y principal de las GPU era la de procesar la imagen que iba a ser mostrada al usuario. Para facilitar esto, se crearon los shaders, pequeños programas gráficos destinados

a ejecutarse en la GPU como parte de la pipeline principal de renderizado, cuya base es transformar inputs en outputs que finalmente formarán la imagen a mostrar.

El pipeline principal de renderizado esta compuesto de manera general por las siguientes etapas: El shader de vértices recibe el input directamente de la CPU, y su trabajo principal es calcular cual será la posición final de cada vertice en la escena. Le sigue la etapa de teselación, la cual es opcional, y cuya labor es la de transformar las superficies formadas por los vértices en primitivas mas pequeñas, aumentando el nivel de detalle de la malla a mostrar. El shader de geometría, también opcional, genera una o más primitivas como output a partir de una primitiva recibida como input. Le siguen las etapas de post-procesado, donde se descarta el area de volúmenes fuera del volumen visible, el ensamblado de primitivas, que ordena la geometría de la escena en una secuencia de primitivas simples (lineas, puntos o triangulos), la rasterización, donde se transforma las primitivas recibidas en fragmentos a traves de los cuales podemos determinar el estado final de cada pixel, y por último el shader de fragmentos, que determina el color final de cada «fragmento» o pixel de la escena. Opcionalmente existen ciertos tests que se ejcutan al terminar el shader de fragmentos.[19]

Durante muchos años, los programadores utilizaban los shaders modificables , es decir, el shader de vertices,de geometría y de fragmentos para realizar ciertas operaciones que no implicaban de manera directa al pipeline gráfico, para así aprovechar la potencia de cómputo que ofrece la GPU. Por este motivo, se introdujeron los compute shaders, shaders que pueden ser ejecutados fuera del pipeline gráfico.[20]

La manera de trabajar con ellos funciona de manera similar a lo explicado anteriormente, cada compute shader ejecuta una funcion que se subdivide en work groups de una, dos o tres dimensiones que a su vez contienen un número de hilos que puede estar subdividido también en hasta 3 dimensiones.

4 Plug-ins y lenguajes de scripting

El desarrollo de software puede llegar a ser complejo. Mientras más funcionalidad se requiere más complejo es mantener el código. Debido a esto, los ingenieros de software han desarrollado una serie de técnicas y metodologías para facilitar el desarrollo de software. Un problema habitual, es querer usar cierta funcionalidad en diferentes programas. Podría implementarse en cada uno de estos, pero en dicho caso se está duplicando código. Además de que hay que mantener la misma funcionalidad en diferentes lugares. Para paliar este problema existen las librerías dinámicas y estáticas [21]. Una librería dinámica no forma parte del programa en sí, es un archivo separado del ejecutable principal. En cambio, una librería estática es parte del ejecutable. Esto tiene varias implicaciones, de las cuales una es particularmente interesante: Se puede cambiar una librería dinámica por otra, actualizando así el programa sin tener que compilarlo entero.

Esta particularidad da lugar a la posibilidad de crear plugins [22]. Sin embargo, un plugin no tiene por qué ser una librería dinámica. Un plugin es cualquier componente de software que permita añadir funcionalidad a un programa sin tener que modificar el código fuente del programa.

Tras investigar distintos mecanismos para extender un programa, se llegó a la conclusión de que hay 3 posibilidades principales. Cada una con sus ventajas y desventajas:

4.1 Extensión mediante librería dinámica

Extender un programa mediante librerías dinámicas es algo habitual. Las librerías dinámicas tienen dos ussos fundamentales. Por un lado, permiten utilizar código común en diferentes programas. Por otro lado, permiten extender un programa sin tener que modificar el código fuente del programa. Además de esta modularidad, también simplifica el desarrollo, pues al separar el código en módulos, solo es necesario compilar aquellos que se han modificado. Un ejemplo claro son las “assemblies” en .NET. [23]. Por último, una ventaja utilizada en el mundo de los videojuegos es que las DLLs en el caso de windows, además de código, permiten guardar recursos como texturas, modelos 3D, sonidos, etc. [24]. Esto permite, por ejemplo, compartir mods de un juego en un solo archivo.

Sin embargo, las librerías dinámicas tienen algunos problema o inconveniencias a tener en cuenta. Por un lado, la compatibilidad entre plataformas. Cada sistema operativo gestiona las librerías dinámicas de una forma disinta [21], en Windows se usan DLLs, en distribuciones Linux se usan SOs, en MacOS se usan dylibs. Por otro lado, la compatibilidad entre versiones. Existen también problemas de seguridad, pues una librería dinámica puede ser modificada por un atacante para lograr la ejecución de código malicioso cierto grado de acceso al sistema.

Por último, un gran problema es la compatibilidad entre distintos lenguajes o versiones del mismo lenguaje [25]. Podemos tener un programa principal que carga una DLL y llama a una función de esta. Si el programa y la DLL están compilados con la misma versión de g++, funcionará, sin embargo, si compilamos la DLL o el programa con clang o msvc podría no funcionar correctamente. Existe una solución para esto que es común a la mayoría de lenguajes, y es la creación de una interfaz en C. El problema de esto es que que al trabajar con el ABI de C no es posible acceder a ciertas características del lenguaje en el que se está trabajando, o en ciertos casos es incluso necesario modificar tu propio programa para que sea compaible [26]. En

el caso de querer usar tipos del lenguaje, es necesario pasar un puntero opaco void* y castearlo, según el lenguaje esto podría suponer un overhead.

4.2 Extensión mediante un archivo propietario

Esta opción consiste en usar una especie de archivo de configuración definida por el desarrollador. El programa leerá este archivo y realizará ciertas acciones. Esto es independiente de la plataforma, no conlleva riesgos de seguridad* y no depende del ABI de ningún lenguaje. Además, al ser un fichero de texto, no es necesario compilarlo. Sin embargo, esta opción es mucho más limitada, pues solo podrá extenderse cierta funcionalidad del programa de forma limitada. Esto puede ser útil para tareas específicas, por ejemplo un sistema danmaku:

```
Danmaku
fire circle-blue <0,-5>
  repeat-every 8
    randomize-x -6 6
        fire-particle
            cartesian-velocity
                randomize-time 0 4
                    periodize 4
                        lerpfromtoback 0.5 1.5 2.5 3.5
                            <0,0.4>
                            <0,0.7>
```

El usuario que consuma esta API está limitado al formato y funciones que el desarrollador ha dado. Además, en sistemas más complejos como el de este ejemplo, esta solución puede ser más difícil de implementar de manera eficiente. Este método de extensión va más alla de un simple archivo de configuración para un sistema con unos parámetros fijos. Esta clase de sistemas puede permitir elegir cierta combinación de parámetros en cierto orden. Este sistema es similar al siguiente que veremos.

4.3 Extensión mediante un lenguaje de scripting

Hay un punto intermedio entre ambas opciones. El uso de DLLs brinda una flexibilidad notable, pero su portabilidad hacia otras plataformas puede ser problemática. Por otro lado, un archivo de configuración es más limitado, requiriendo que el desarrollador implemente un parseador (si utiliza un formato propio) y luego genere las funciones adecuadas. Dar el paso hacia un lenguaje de scripting va un poco más allá. Aquí, el programador se libera de la necesidad de crear un parseador, ya que el intérprete del lenguaje asume esta tarea. Este intérprete puede interactuar con el programa principal a través de una API, permitiendo al programador aprovechar todas las funciones que el lenguaje de scripting ofrece. Además, al igual que los archivos de configuración, este método es multiplataforma, ya que los intérpretes suelen estar disponibles en distintos sistemas operativos. No obstante, toda esta flexibilidad conlleva problemas de seguridad. Además, los lenguajes interpretados suelen ser más lentos que los compilados, lo cual debe tenerse en cuenta especialmente si se prioriza el rendimiento o si el código del lenguaje de scripting se encuentra en un camino crítico de ejecución (hotpath). Se debe tener en cuenta también el overhead por transferir datos de un lenguaje a otro, en el caso de Lua esto implica modificar constantemente un stack. Para demostrar esto, a continuación se muestra una pequeña prueba comparando la ejecución de código en C++ desde una DLL y desde un script en LuaJit usando Sol2 como bridge. Las pruebas se ejecutan en release.

Entorno común:

Por ahora pongo esto aquí, pero no sé si debería poner el resultado y poner esta explicación y código como un fichero anexo. Tengo muchos más benchmarks, este es simplemente el más significativo.

Se define una estructura de la siguiente forma:

```C++
struct ship {
  int life = 100;

  bool hurt(int by) {
    life -= by;
    return life < 1;
  }
};
```

Para el test de lua se creará obtendrá una función de lua que usará nuestro struct de C++ y se llamará desde C++ 10000*10000 veces. Para exponer la función a Lua se usará la api c_call de sol2. Esta es menos legible que el api común de sol2, pero es más rápida.*

Además de que la doc de sol2 menciona que esta forma es más aunque sea más lenta, hice pruebas, no las puse porque no creo que sean importantes. Pero por poder podría hasta explicar porque esto es más rápido que lo otro, pero creo que se sale del ámbito de este TFG.

```C++
  sol::state lua;
  lua.open_libraries(sol::lib::base);
  lua.set("shiphurt", sol::c_call<decltype(&ship::hurt), &ship::hurt>);
```

Finalmente, antes de ejecutar la prueba, se creará una instancia de ship con 2000000 de vida y se pasará a Lua. Además, se creará una función en la tabla global y la guardaremos en C++ en un std:function.

```C++
ship s;
s.life = 2000000;

lua.set("ship", &s);

const auto& code = R"(
  _G.testfunc = function(amount) shiphurt(ship, amount)  end
)";

lua.script(code); // Ejecutar el código de Lua

sol::function func = lua["testfunc"]; // Guardar la función de lua en C++
```

Finalmente, se ejecutará la función de Lua 10000*10000 veces.

```C++
auto start = std::chrono::high_resolution_clock::now();

const int size = 10000;
const int amount = 20;

sol::function func = lua["testfunc"];
for (int i = 0; i < size * size; i++)
{
```

```
9     func(amount);
10 }
11
12 std::cout << "ship life is: " << s.life << std::endl;
13
14 auto end = std::chrono::high_resolution_clock::now();
15 auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end -
   start);
16
17 std::cout << "Time taken: " << duration.count() << " miliseconds" << std::endl;
```

El output obtenido:

```
ship life is: -1998000000
Time taken: 4959 miliseconds
```

En C++ la configuración es similar, pero se usa Dylib [27] para cargar la librería dinámica y obtener la función.

```
1   dylib lib("CDll", dylib::extension);                                    C++
2   auto func = lib.get_function<bool(ship&, int)>("testfunc");
3
4   [...]
5
6   for (int i = 0; i < size * size; i++)
7   {
8     func(s, amount);
9   }
```

El output obtenido:

```
ship life is: -1998000000
Time taken: 167 miliseconds
```

Tengo otro test pero a la inversa, una sola llamada a lua o a la dll, y es lua/dll la que realiza un montón de iteraciones, esa prueba la hice para ver mejor el overhead, porque no es lo mismo llamar 100000000 a Lua que llamar una vez y que Lua haga 100000000 operaciones. Al fin y al cabo te ahorras 1000000 * 3 modificaiones del stack (una para poner la función al top, otra para poner el parámetro y otra para el de retorno, que aunuqe no se use se pone en el stack). Y esto funciona así porque también hice este test sin usar Sol como librería, lo hice también usando lua a pelo modificando el stack, pero la diferencia era inexistente así que para el ejemplo usé Sol que es más legible.

Este texto a continuación puede ser redundante, pero es como estoy enfocando el TFG ahora, una comparación entre implementar el sistema de partículas de Noita en CPU y GPU. ¿Quié tiene que ver el sistema de scripting en esto? Que Noita usa Lua pero como configurador para varitas y pequeños comportamientos (la simulación es en C++ y no se puede tocar hasta donde sé, pero creo que algú mod lo hace), por lo tanto este sistema debe ser tanto eficinete como modeable, en el TFG intentar´eexplicarlo mejor para que sea coherente porque es la única forma de enfocarlo que se me ocurre.

Esto demuestra la diferencia que puede llegar a haber entre ambas aproximaciones a un sistema modular que aproveche la CPU. Como el objetivo es recrear el sistema de partículas de Noita explorando todas las posibilidades de la CPU, se implementarán los sistemas de extensión

mediante librerías dinámicas y lenguajes de scripting. Tras esto, se valorán ambas opciones, comparándolas con su alternativa en GPU para así poder determinar cuál es la mejor opción para este caso en concreto.

Quizás pienses que he hablado poco de Lua, y es cierto, me estoy reservando el resto de destalles de Lua para la parte de implementación en CPU en Lua.

5 Simulador en CPU

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.

5.1 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.

5.1.1 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;
  std::vector<Interaction> interactions;
};
```

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.

5.1.2 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.

5.2 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.

5.2.1 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.

5.2.2 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.

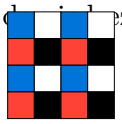

Figura 11: Patrón de ajedrez

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.

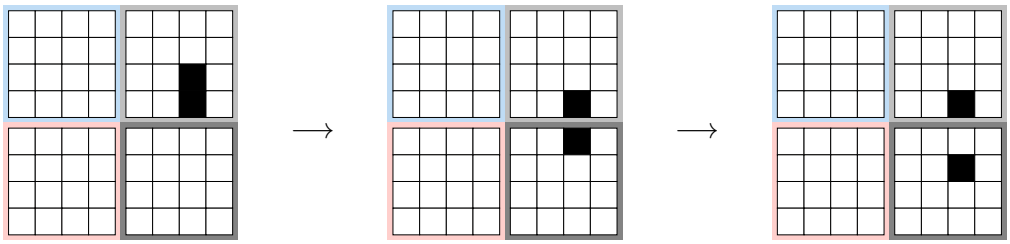

Figura 12: Problema de multithreading

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

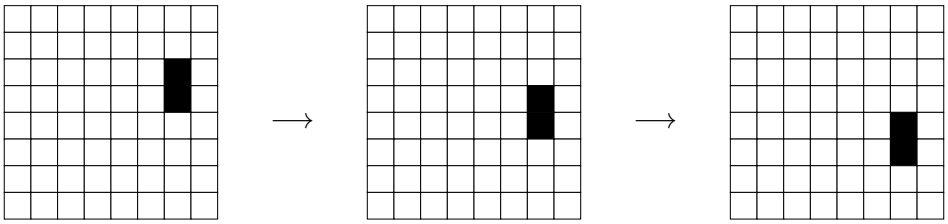

Figura 13: Resultado esperado

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.

5.3 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.

6 Simulador en GPU

7 Comparación y pruebas

8 Conclusiones y trabajo futuro

Bibliografía

[1] «MATHEMATICAL GAMES. The fantastic combinations of John Conway\'s new solitaire game 'life'». [En línea]. Disponible en: https://www.ibiblio.org/lifepatterns/october1970.html

[2] Thomas M. Li, *Cellular Automata*. 2010.

[3] Andrew Adamatzky, *Game of Life Cellular Automata*, vol. 1. 2010.

[4] Karl-Peter Hadeler y Johannes Müller, *Cellular Automata':' Analysis and Applications*, vol. 1. 2017.

[5] «Rule 30 of Wolfram Automata». [En línea]. Disponible en: https://mathworld.wolfram.com/Rule30.html

[6] Wikipedia, «Autómata Celular». [En línea]. Disponible en: https://es.wikipedia.org/wiki/Aut%C3%B3mata_celular

[7] Wikipedia, «Computational fluid dynamics». [En línea]. Disponible en: https://en.wikipedia.org/wiki/Computational_fluid_dynamics

[8] Wikipedia, «Particle System». [En línea]. Disponible en: https://en.wikipedia.org/wiki/Particle_system

[9] Wikipedia, «Sand Simulator». [En línea]. Disponible en: https://en.wikipedia.org/wiki/Falling-sand_game

[10] Web Archive, «Falling Sand Game». [En línea]. Disponible en: https://web.archive.org/web/20090423105358/http://fallingsandgame.com/overview/index.html

[11] Powder Toy, «Powder Toy». [En línea]. Disponible en: https://powdertoy.co.uk/

[12] Max Bittker, «Sandspiel». [En línea]. Disponible en: https://maxbittker.com/making-sandspiel

[13] Max Bittker, «Sandspiel Club». [En línea]. Disponible en: https://sandspiel.club/

[14] Google, «Blockly». [En línea]. Disponible en: https://developers.google.com/blockly

[15] Jon Peddie, *The History of the GPU - Steps to Invention*, vol. 1. 2023.

[16] «Nvidias Geforce 256». [En línea]. Disponible en: https://www.computer.org/publications/tech-news/chasing-pixels/nvidias-geforce-256

[17] Jon Peddie, *Professional CUDA C Programming*, vol. 1. 2014. [En línea]. Disponible en: https://www.cs.utexas.edu/~rossbach/cs380p/papers/cuda-programming.pdf

[18] «Cuda C++ Programming Guide». [En línea]. Disponible en: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[19] «Rendering Pipeline Overview». [En línea]. Disponible en: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

[20] «Compute Shaders Introduction». [En línea]. Disponible en: https://learnopengl.com/Guest-Articles/2022/Compute-Shaders/Introduction

[21] John R. Levine, *Linkers and Loaders*.

[22] Lázaro Bustio-Martínez, Yenice Coma Peña, y Isneri Talavera Bustamante, *Arquitectura basada en plugins para el desarrollo de software científico*. 2013.

[23] «Assemblies in .NET». [En línea]. Disponible en: https://learn.microsoft.com/en-us/dotnet/standard/assembly/

[24] «Creating a resource-only DLL». [En línea]. Disponible en: https://learn.microsoft.com/en-us/cpp/build/creating-a-resource-only-dll?view=msvc-170

[25] «Understanding Application Binary Interface (ABI)». [En línea]. Disponible en: https://link.springer.com/content/pdf/10.1007/978-1-4612-3192-9_25.pdf

[26] «The Rustonomicon: Alternative representations». [En línea]. Disponible en: https://doc.rust-lang.org/nomicon/other-reprs.html

[27] «Dylib - Dynamic Library Loader for C++». [En línea]. Disponible en: https://github.com/Layty/Dylib