Trabajo de Fin de Grado

Grado en Desarrollo de Videojuegos

Exploración y análisis de rendimiento y extensibilidad de diferentes implementaciones de simuladores de partículas

Exploration and analysis of performance and extensibility of different implementations of particle simulators.

Universidad Complutense de Madrid

Alumnos

Nicolás Rosa Caballero
Jonathan Andrade Gordillo

Dirección

Pedro Pablo Gómez Martin

7 de mayo de 2024

1 Agradecimientos

Nicolás

Quiero dar las gracias a mis padres, por su paciencia y comprensión cuando estaba agobiado, y a mi hermano, por su apoyo y ánimos. A mis amigos, por su apoyo y por estar siempre ahí cuando los necesito. A mi compañero de proyecto, por su colaboración y esfuerzo. A mi tutor, por su ayuda y orientación. A todos ellos, gracias.

Jonathan

Quiero dar las gracias, de igual manera, a mis padres por su incondicional apoyo, a mis amigos, por estar ahí para lo bueno y lo malo, a Marta, por haberme dado fuerzas incluso cuando no las tenía ella, a mi compañero de proyecto por ser una inspiración como desarrollador de software y por su gran trabajo y, por último y no menos importante, a mi tutor por su orientación y por todas los momentos y reuniones que hemos tenido durante el TFG y durante la carrera.

2 Resumen

Los simuladores de arena, subgénero de autómatas celulares, han experimentado un resurgimiento en popularidad recientemente. Sin embargo, hemos identificado un obstáculo significativo que dificulta su adopción más generalizada tanto entre usuarios como desarrolladores, y este es la escasez de antecedentes o ejemplos disponibles. Por lo tanto, el objetivo de este proyecto es investigar diversas implementaciones de estos simuladores, evaluando sus ventajas e inconvenientes, así como su capacidad para ser ampliados por cualquier usuario, con el fin de aumentar la visibilidad y comprensión de este subgénero.

Para lograr este objetivo, el proyecto examinará varias implementaciones, tanto en términos de su ejecución en la CPU como de una versión que ejecute la lógica en la GPU. Además, se llevará a cabo un estudio con usuarios reales para identificar posibles problemas con las implementaciones y evaluar el interés general en el proyecto. Los resultados de estos análisis se utilizarán para extraer conclusiones y proponer posibles mejoras para el proyecto.

3 Palabras clave

- Simuladores de arena
- Automatas celulares
- Programación paralela
- Multihilo
- GPU
- Lua
- Rust
- WebAssembly
- Blockly
- Compute shader

4 Abstract

Sand simulators, a subgenre of cellular automata, have experienced a resurgence in popularity in recent years.However, we have identified a significant barrier that hinders its broader adoption among both users and developers, and this is the scarcity of background or available examples. Therefore, the aim of this project is to investigate various implementations of these simulators, evaluating their advantages and disadvantages, as well as their ability to be extended by any user, in order to increase the visibility and understanding of this subgenre.

To achieve this goal, the project will examine several implementations, both in terms of their execution on the CPU and a version that runs the logic on the GPU. Additionally, a study will be conducted with real users to identify potential issues with the implementations and assess the overall interest in the project. The results of these analyses will be used to draw conclusions and propose possible improvements for the project.

5 Key Words

- Sand simulators
- Cellular automata
- Parallel programming
- Multithreading
- GPU
- Lua
- Rust
- WebAssembly
- Blockly
- Compute shader

Índice

Índice de Figuras

6 Introducción

6.1 Motivación

Los simuladores de arena fueron un subgénero emergente durante la década de los noventa, y continuaron siendo populares hasta principios de los años 2000. Durante ese tiempo, surgieron muchos programas y juegos que permitían a la gente interactuar con mundos virtuales llenos de partículas simuladas. Esto atrajo tanto a amantes de la simulación como a desarrolladores de videojuegos. Sin embargo, tras un período de relativa tranquilidad, estamos presenciando un nuevo auge del género de la mano de videojuegos como Noita o simuladores sandbox como Sandspiel.

Este proyecto nace del deseo de sumergirnos en el mundo de los simuladores de arena. Queremos explorar sus diferentes aspectos y características así como entender mejor las ventajas y desventajas de diferentes enfoques de desarrollo de cara al usuario, para así poder contribuir a su evolución y expansión, ya que consideremos que es un subgénero que puede dar muy buenas experiencias de juego y de uso.

En resumen, queremos entender y ayudar a mejorar los simuladores de arena para hacerlos más útiles y efectivos para los usuarios.

6.2 Objetivos

El principal objetivo de este TFG es estudiar el comportamiento y aprendizaje de usuarios haciendo uso de diferentes implementaciones de simuladores de arena.

Se valorará la funcionalidad de cada implementación haciendo uso de los siguientes parámetros:

- Comparación de rendimiento: se compararán bajo las mismas condiciones, tanto a nivel de hardware como en uso de partículas midiendo el rendimiento final conseguido. Este rendimiento se comparará haciendo uso de razón nº partículas / frames por segundo conseguidos. Idealmente se averiguará la mayor cantidad de partículas que cada simulador puede soportar manteniendo 60 fps.
- Comparación de usabilidad: se estudiará el comportamiento de un grupo de usuarios para valorar la facilidad de uso y de entendimiento de sistemas de ampliación de los sistemas que permitan expansión por parte del usuario. Se valorará la rapidez para realizar un set de tareas asi como los posibles desentendimientos que puedan tener a la hora de usar el sistema.

Con estos análisis, se pretende explorar las características que contribuyen a una experiencia de usuario óptima en un simulador de arena, tanto en términos de facilidad de uso como de rendimiento esperado por parte del sistema. Al comprender mejor estas características, se podrán identificar áreas de mejora y desarrollar recomendaciones para optimizar la experiencia general del usuario con los simuladores de arena.

Nuestro objetivo es encontrar un balance entre rendimiento y facilidad de extensión que proporcione tanto un entorno lúdico a usuarios casuales como una base sólida de desarrollo para desarrolladores interesados en los simuladores de arena.

6.3 Plan de trabajo

La planificación de trabajo se realizó mayormente entre los autores del TFG, apoyandonos en nuestro tutor mediante reuniones periódicas para ayudarnos a medir nuestro ritmo de trabajo. Al comienzo del proyecto se definieron una serie de tareas necesarias para considerar al desarrollo exitoso. Estas tareas se planificaron en este orden:

- Investigación preliminar sobre los conceptos fundamentales de los autómatas celulares y los simuladores de arena, así como decidir y discutir qué librerias y que software se utilizará.
- Implementación del Simulador en C++, haciendo uso de OpenGL y GLFW: el objetivo de este simulador era tener una base referencial sobre la que apoyarnos a la hora de desarrollar el resto de simuladores, además de permitirnos aplicar de manera un poco más laxa los conocimientos aprendidos en la fase preliminar de investigación sobre autómatas celulares.
- Desarrollo de Simulador en LUA con LÖVE: el objetivo de esta implementación era aplicar la funcionalidad del sistema anterior añadiendo capacidad de adición de parículas personalizadas por parte del usuario, además de añadir capacidad de multithreading para mejorar el rendimiento.
- Desarrollo de Simulador con ejecución de lógica mediante GPU: el objetivo de esta implementación era tener una referencia a nivel de rendimiento sobre la que comparar el resto de implementaciones, así como explorar la viabilidad de programar el sistema al completo mediante GPU.
- Desarrollo de Simulador en Rust haciendo uso de Macroquad y Blockly para la creación de bloques personalizados: el objetivo de esta implementación era explorar la opción de crear una version cuya accesibilidad mayor para un perfil de usuario no técnico,además de un rendimiento potencialmente superior en comparación a la implementación de LUA.
- Realización de pruebas de usuario: comparar mediante los parámetros mencionados anteriormente 6.2 .
- Análisis de los datos y comparativa entre los resultados obtenidos por las diferentes implementaciones.
- Conclusiones y trabajo futuro: a partir del análisis de los datos previos, se extraen conclusiones y se explora el potencial para futuras expansiones del proyecto.

7 Introduction

7.1 Motivation

Sand simulators were a prominent subgenre during the 1990s, and remained popular until the early 2000s. During that time, many programs and games emerged that allowed people to interact with virtual worlds filled with simulated particles. This attracted both simulation lovers and video game developers. However, after a period of relative calm, we are witnessing a renaissance of the genre thanks to video games like Noita or sandbox simulators like Sandspiel.

This project comes from the desire to immerse ourselves in the world of sand simulators. We want to explore its different aspects and characteristics as well as better understand the advantages and disadvantages of different development approaches for the user, in order to contribute to its evolution and expansion, since we consider that it is a subgenre that can provide very good gaming and user experiences.

In short, we want to understand and help improve arena simulators to make them more useful and effective for users.

7.2 Objectives

The main objective of this TFG is to study the behavior and ease of learning of different users using different implementations of sand simulators.

The functionality of each implementation will be assessed using the following parameters:

- Performance comparison: they will be compared under the same conditions, both at the hardware level and in the use of particles, measuring the final performance achieved. This performance will be compared using the ratio of number of particles / frames per second achieved. Ideally, you will find out the largest number of particles that each simulator can support while maintaining 60 fps.
- Comparison of usability: the behavior of a group of users will be studied to assess the ease of use and understanding of expansion systems that allow expansion by the user. The speed of completing a set of tasks will be assessed, as well as any possible misunderstandings they may have when using the system.

With these analyses, we aim to explore the characteristics that contribute to an optimal user experience in an arena simulator, both in terms of ease of use and expected performance of the system. By better understanding these characteristics, areas for improvement can be identified and recommendations developed to optimize the overall user experience with arena simulators.

Our goal is to find a balance between performance and extensibility that provides both a playful environment for casual users and a development base for developers interested in arena simulators.

7.3 Work plan

Work planning was mostly carried out between the authors of the TFG, relying on our tutor through regular meetings to help us measure our work pace. At the beginning of the project, a series of tasks were defined as necessary for successful development. These tasks were planned in this order:

- Preliminary research on the fundamental concepts of cellular automata and sand simulators, as well as deciding and discussing which libraries and software will be used.
- Implementation of the Simulator in C++, using OpenGL and GLFW: the aim of this simulator was to have a reference base to support us when developing the rest of the simulators, as well as allowing us to apply in a slightly more lax way the knowledge learned in the preliminary research phase on cellular automata.
- Development of Simulator in LUA with LÖVE: the aim of this implementation was to apply the functionality of the previous system by adding the ability for the user to add custom particles, as well as adding multithreading capability to improve performance.
- Development of Simulator with logic execution through GPU: the aim of this implementation was to have a performance reference to compare the rest of the implementations, as well as to explore the viability of programming the entire system through GPU.
- Development of Simulator in Rust using Macroquad and Blockly for the creation of custom blocks: the aim of this implementation was to explore the option of creating a version with greater accessibility for a non-technical user profile, as well as potentially superior performance compared to the LUA implementation.
- User testing: compare using the parameters mentioned above 7.2.
- Analysis of the data and comparison between the results obtained by the different implementations.
- Conclusions and future work: from the analysis of the previous data, conclusions are drawn and the potential for future project expansions is explored.

8 Autómatas celulares y simuladores de arena

En este capítulo se hablará sobre el concepto de automátas celulares, su historia y su relevancia en la ciencia y matemáticas. Además, se presentarán algunos ejemplos de autómatas celulares relevantes en la investigación científica. Posteriormente se hablará sobre los simuladores de arena, su impacto y su relación con los autómatas celulares.

8.1 Autómatas celulares

Los autómatas celulares [3] son un modelo matemático discreto que se representa como una matriz n-dimensional de celdas. Este modelo puede ser de cualquier número de dimensiones, aunque los más comunes son los autómatas celulares de una y dos dimensiones. Cada celda en esta matriz puede existir en uno de varios estados posibles, que son finitos en número. Existe un consenso sobre la definición teórica de automátas celulares que los describe como modelos deterministas, sin embargo, en su aplicación práctica, los autómatas celulares pueden ser tanto deterministas como estocásticos. Esto es debido a que la naturaleza de ciertos sistemas dependen de variables aleatorias, por lo que resulta más adecuado modelarlos como sistemas estocásticos.

El cambio de estado de una celda se determina en función de los estados de las celdas vecinas y se produce en unidades discretas de tiempo, denominadas generaciones [4]. En este contexto, el término «vecinos» hace referencia a las celdas que se encuentran adyacentes a una celda específica.

En el caso particular de los autómatas celulares de dos dimensiones, existen dos tipos de vecindades que se utilizan comúnmente: la vecindad de Moore y la vecindad de Neumann [5]. En la vecindad de Moore, se consideran como vecinos las ocho celdas que rodean a una celda central, incluyendo las celdas diagonales. En la vecindad de Neumann, por otro lado, solo se consideran como vecinos las cuatro celdas que están directamente arriba, abajo, a la izquierda y a la derecha de la celda central.

Finalmente, para definir completamente un autómata celular, se requiere una «regla» que se aplica a cada celda. Esta regla establece cómo cambiará el estado de una celda en la siguiente generación, basándose en los estados actuales de la celda y sus vecinos. Dicho de otro modo, la regla es la que transforma el sistema de celdas de su estado actual al estado siguiente en cada generación.

Para ilustrar mejor el concepto de autómata celular se muestra un ejemplo a continuación. Este es un automáta celular de 2 dimensiones. Cada celda puede tener dos posible estados: `Apagada` o `Encendida`. La regla de este sistema es que si la celda está `Apagada` y cualquier de sus vecinos está `Encendida` entonces pasa a estar `Encendida` en la siguiente iteración. Por el contrario, si la celda está `Encendida` y cualquiera de sus vecinos está `Apagado` entonces pasará al estado `Apagado`. En resumen, los estados alternan periódicamente.

A continuación se muestra un ejemplo de este sistema con 3 iteraciones:

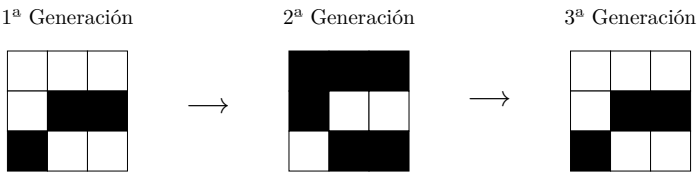


Figura 1: Ejemplo de autómata celular sencillo

Una característica distintiva de los autómatas celulares es su evolución «simultánea» o en bloque. En este modelo matemático, todas las celdas cambian de estado al mismo tiempo, de una generación a la siguiente. Esta simultaneidad en la evolución es fundamental para la dinámica del autómata celular, contribuyendo a su robustez y predictibilidad.

Muchos automátas celulares pueden representarse como una máquina de Turing [6]. Una máquina de Turing es un modelo matemático de un dispositivo que manipula símbolos en una cinta de acuerdo con una serie de reglas. Aunque las máquinas de Turing son abstractas y teóricas, se consideran un modelo fundamental de la computación y han sido utilizadas para demostrar la existencia de problemas no computables.

A su vez, la Máquina de Turing puede ser vista como un autómata que ejecuta un procedimiento efectivo formalmente definido. Este procedimiento se lleva a cabo en un espacio de memoria de trabajo que, teóricamente, es ilimitado. A su vez, algunos autómatas celulares pueden ser vistos como máquinas de Turing.

A continuación, se procede a examinar su trayectoria histórica y su significativa contribución a la ciencia y las matemáticas.

8.1.1 Historia

El concepto de autómatas celulares tiene sus raíces en las investigaciones llevadas a cabo por John von Neumann en la década de 1940. Von Neumann fue un matemático húngaro-estadounidense, principalmente reconocido por la arquitectura de computadoras que lleva su nombre. Además de esto, realizó muchas contribuciones en diversos campos: Participó en el proyecto Manhattan, contribuyó de forma notable a la teoría de juegos, a la teoría de conjuntos y en física cuántica. Además de esto realizó importantes contribuciones en la teoría de autómatas y sistemas auto-replicantes [7].

Von Neumann se enfrentó al desafío de diseñar sistemas auto-replicantes, y propuso un diseño basado en la autoconstrucción de robots. Sin embargo, esta idea se encontró con obstáculos debido a la complejidad inherente de proporcionar al robot un conjunto suficientemente amplio de piezas para su replicación. A pesar de estos desafíos, el trabajo de von Neumann en este campo fue fundamental y marcó un punto de inflexión significativo. Su trabajo seminal en este campo fue documentado en un artículo científico de 1948 titulado «The general and logical theory of automata» [8].

Stanislaw Ulam, un matemático polaco y colega de von Neumann, conocido por su trabajo en la teoría de números y su contribución al diseño de la bomba de hidrógeno, propuso durante este período una solución alternativa al desafío de la auto-replicación. Ulam sugirió la utilización

de un sistema discreto para crear un modelo reduccionista de auto-replicación. Su enfoque proporcionó una perspectiva que ayudó a avanzar en el campo de los autómatas celulares.

En 1950, Ulam y von Neumann desarrollaron un método para calcular el movimiento de los líquidos, concebido como un conjunto de unidades discretas cuyo movimiento se determinaba según los comportamientos de las unidades adyacentes. Este enfoque innovador sentó las bases para el nacimiento del primer autómata celular conocido.

Los descubrimientos de von Neumann y Ulam propiciaron que otros matemáticos e investigadores 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 [9] en la revista Scientific American sobre un automáta celular específico: el juego de la vida de Conway. Su artículo generó un gran interés en los autómatas celulares y los sistemas dinámicos, lo que llevó a un aumento significativo en la investigación y experimentación en este campo. En el siguiente apartado se hablará en más detalle sobre el juego de la vida de Conway y otros autómatas celulares relevantes.

8.1.2 Ejemplos

A continuación se mostrarán diversos ejemplos de automátas celulares, comenzando por el conocido `Juego de la Vida de Conway` [4]. Posteriormente, se presentarán: los autómatas de Wolfram, la Hormiga de Langton, el autómata de contacto y el autómata de Greenberg-Hastings.

8.1.2.1 Juego de la vida

El Juego de la Vida, concebido por el matemático británico John Horton Conway, es un autómata celular bidimensional que se desarrolla en una cuadrícula teóricamente infinita de células cuadradas. Cada una de estas células puede estar en uno de dos posibles estados: `viva` (negra) o `muerta` (blanca).

El Juego de la Vida es un ejemplo de un sistema que exhibe comportamiento complejo a partir de reglas simples. Estas reglas, que determinan el estado de una célula en la siguiente generación, son las siguientes:

- Si una célula viva está rodeada por más de tres células vivas, muere por sobrepoblación.
- Si una célula viva está rodeada por dos o tres células vivas, sobrevive.
- Si una célula viva está rodeada por menos de dos células vivas, muere por soledad.
- Si una célula muerta está rodeada por exactamente tres células vivas, se vuelve viva.

Para el conteo de células vivas adyacentes se considera una vencidad de Moore — es decir, se tienen en cuenta las ocho células que rodean a una célula central.

Como se mencionó anteriomente, estas reglas se aplican simultáneamente a todas las células en la cuadrícula — es decir, todos los cambios de estado ocurren al mismo tiempo en cada paso de tiempo.

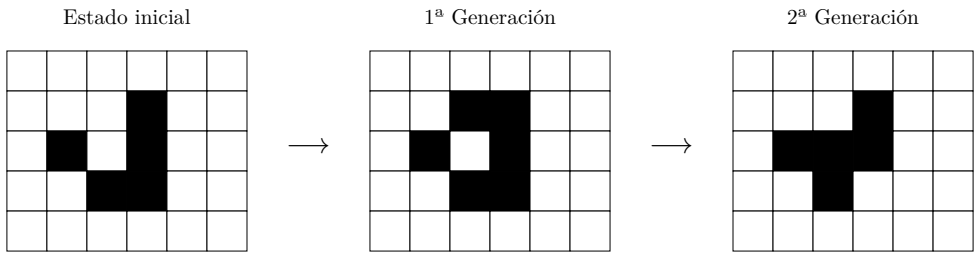


Figura 2: Ejemplo del Juego de la Vida

A pesar de su aparente simplicidad, el Juego de la Vida es capaz de generar una diversidad notable de patrones y estructuras. Algunos de estos patrones son estático. Un patrón estático es aquel en el que las células que lo forman mantienen su estado de generación en generación [4]. Un ejemplo de esto es el «bloque», que consiste en un cuadrado de 2x2 células vivas rodeadas por célular muertas o viceversas, y el «bote», que se asemeja a una forma de L compuesta por cinco células vivas. La Figura 3 muestra ejemplos de estructuras estáticas en el Juego de la Vida.

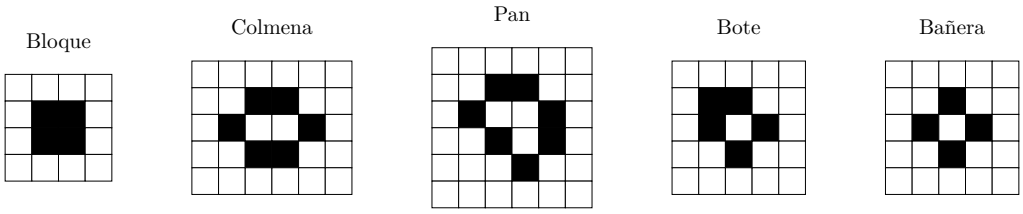


Figura 3: Estructuras estáticas en el juego de la vida

Existen también patrones oscilatorios, que alternan entre dos o más configuraciones. Un ejemplo sencillo de esto es el «blinker», una formación lineal de tres células vivas que oscila entre una orientación horizontal y vertical. La Figura 4 muestra un ejemeplo de estructura oscilatoria en el Juego de la Vida.

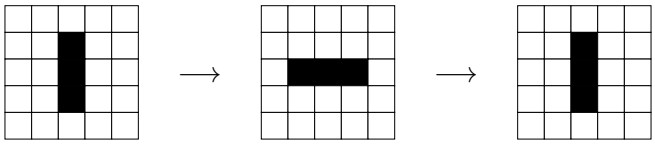


Figura 4: Blinker, estructura oscilatoria del juego de la vida

Adicionalmente, se pueden observar patrones que se desplazan a lo largo de la cuadrícula, a los que se les denomina «naves espaciales». El más simple de estos es el «planeador», un patrón de cinco células que se mueve en una trayectoria diagonal a través de la cuadrícula.

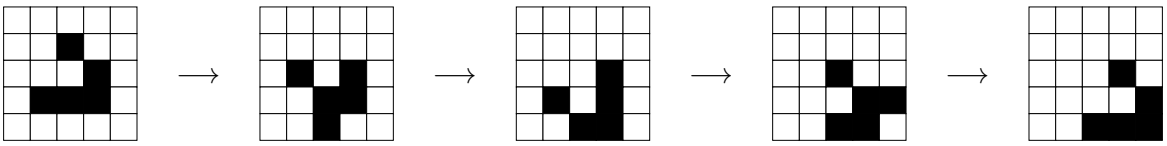


Figura 5: Planeador, estructura moviente del juego de la vida

El Juego de la Vida ha sido objeto de considerable estudio en el campo de la teoría de la complejidad. Ha demostrado ser un modelo útil para explorar conceptos como la autoorganización, la emergencia de la complejidad en sistemas dinámicos, y la computación universal (la capacidad de simular una máquina de Turing) [4], [3]. A pesar de su aparente simplicidad, el Juego de la Vida esconde una riqueza de comportamientos complejos y sorprendentes, y continúa siendo un área activa de investigación y experimentación.

8.1.2.2 Autómatas de Wolfram

Los Autómatas de Wolfram [5], ideados por el físico y matemático Stephen Wolfram, son un conjunto de reglas que rigen el comportamiento de autómatas celulares unidimensionales con estados binarios. Estos autómatas consisten en una línea de celdas, cada una de las cuales puede estar en uno de dos estados: 0 o 1.

Cada regla en el conjunto de autómatas de Wolfram determina cómo cambia el estado de una celda en función de su estado actual y los estados de sus vecinos inmediatos (la celda a la izquierda y la celda a la derecha). Dado que cada celda y sus dos vecinos pueden estar en uno de dos estados, hay $2^3 = 8$ configuraciones posibles para una celda y sus vecinos.

Cada regla se puede representar como un número binario de 8 bits, donde cada bit corresponde a una de las 8 configuraciones posibles de una celda y sus vecinos. Por lo tanto, hay $2^8 = 256$ reglas posibles, numeradas del 0 al 255.

Aunque los autómatas de Wolfram son unidimensionales, a menudo se visualizan en dos dimensiones para mostrar cómo evolucionan con el tiempo. En esta visualización, cada generación (o iteración) del autómata se representa como una nueva fila debajo de la fila anterior. Esto permite ver cómo los estados de las celdas cambian con el tiempo y cómo emergen patrones a partir de las reglas simples del autómata.

A continuación se muestra la regla 30 [10] de Wolfram tras ejecutar 15 iteraciones de este:

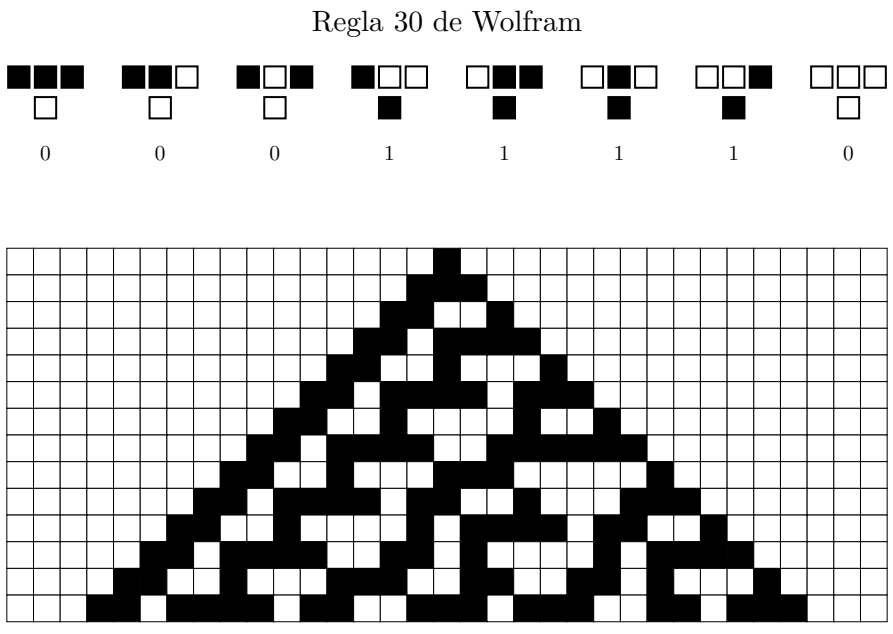


Figura 6: Ejemplo autómata de Wolfram

8.1.2.3 Hormiga de Langton

La «hormiga de Langton» [5], [11], es una máquina de Turing bidimensional de 4 estados, se describe de manera sencilla de la siguiente manera. Se considera un tablero cuadriculado donde cada casilla puede ser negra o blanca, y también puede contener una hormiga. Esta hormiga tiene cuatro direcciones posibles: norte, este, oeste y sur. Su movimiento sigue reglas simples: gira 90 grados a la derecha cuando está sobre una casilla negra, y 90 grados a la izquierda cuando está sobre una casilla blanca, tras lo cual "avanza" en dicha dirección. Además, al "dejar" una casilla, ésta cambia de color. El proceso comienza con una sola hormiga en una casilla blanca. Al principio, su movimiento parece caótico, pero después de un cierto número de pasos, se vuelve predecible, repitiendo un patrón cada cierto tiempo. En este punto, la parte del rastro de la hormiga que está en casillas negras crece de manera periódica, extendiéndose infinitamente por el tablero.

En el autómata celular de la hormiga de Langton, se tienen 10 estados posibles. Estos se derivan de 2 colores de celda (blanco y negro), y la presencia de la hormiga. Cuando la hormiga está ausente, se consideran los 2 estados de color. Cuando la hormiga está presente, se consideran 4 direcciones posibles (norte, este, oeste y sur) para cada color de celda. Por lo tanto, se tienen 2 estados (colores) cuando la hormiga está ausente, y 2 (colores) * 4 (direcciones) = 8 estados cuando la hormiga está presente. En total, se tienen 2 + 8 = 10 estados.

Cabe destacar que la hormiga no se mueve en sí misma, sino que cambia el estado de las celdas en las que se encuentra [5]. Cuando la hormiga está en una celda, esa celda cambia de color y la hormiga desaparece. Sin embargo, una de las celdas adyacentes notará que la celda vecina tenía una hormiga orientada en su dirección, lo que provocará que su estado cambie para incluir la hormiga en la siguiente iteración.

Para una mejor comprensión, se pueden considerar dos celdas adyacentes: una celda blanca sin hormiga y, a su derecha, otra celda blanca con una hormiga orientada hacia arriba. En un autómata celular, el estado de una celda es dependiente de sus vecinos. En este escenario, la celda vacía detecta que su celda vecina contiene una hormiga orientada hacia arriba y es de color blanco. De acuerdo con las reglas de la hormiga de Langton, la hormiga debería girar a la izquierda, que es la ubicación de la celda vacía. Como resultado, el estado de la celda vacía cambiará a ser blanca pero ahora con una hormiga orientada hacia la izquierda. Simultáneamente, la celda que originalmente contenía la hormiga cambiará su estado a estar vacía y se tornará de color negro.

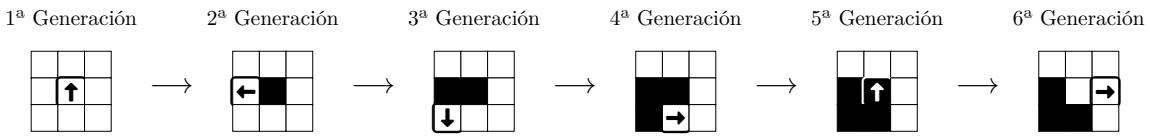


Figura 7: Ejemplo simple de la hormiga de Langton

A partir de las 10000 generaciones aproximadamente, la hormiga de Langton muestra un comportamiento periódico que se repite en un ciclo de 104 generaciones. Este es el resultado en la generación 11000:

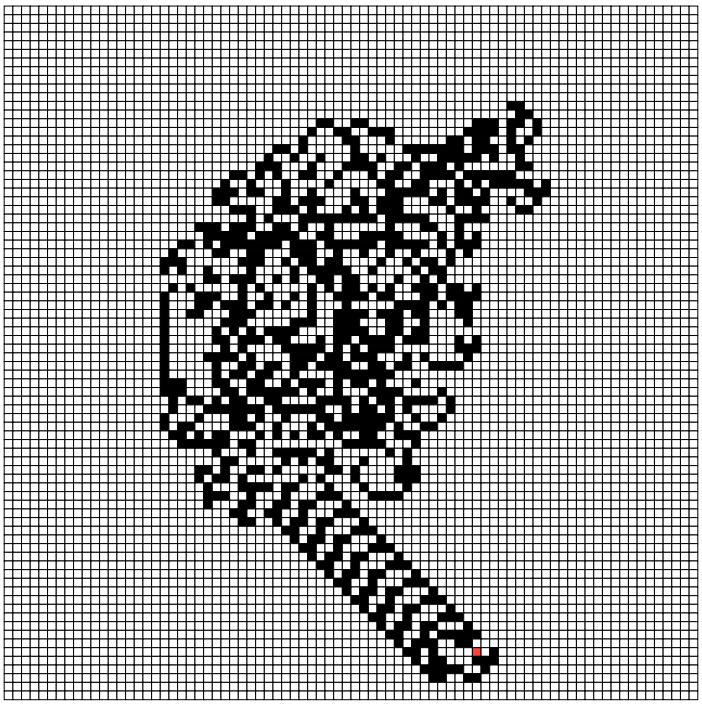


Figura 8: Ejemplo completo de la hormiga de Langton

8.1.2.4 Autómata de Contacto

Un autómata de contacto [5] puede es uno de los modelos más simples de la propagación de una enfermedad infecciosa. Este autómata está compuesto por 2 estados: `celda infectada` (negra) y `celda no infectada` (blanca). Las reglas son las siguiente: Una celda infectada nunca cambia, una celda no infectada se vuelve una celda infectada si cualquiera de las 8 celdas adyacente es una infectada. Existe una versión no determinista en la que la una celda no infectada se infecta pero con una probabilidad P. Este modelo es muy útil en la investigación de la propagación de enfermedades infecciosas, ya que en una situación real existen variables aleatorias como se mencionó anteriormente.

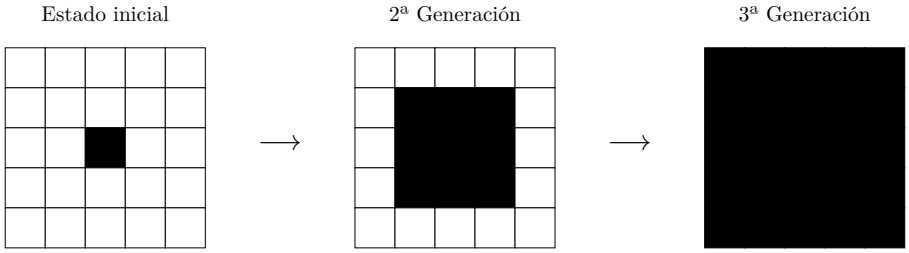


Figura 9: Ejemplo de autómata de contacto determinista

8.1.2.5 Autómata de Greenberg-Hastings

Los autómatas de Greenberg-Hastings [5] son modelos bidimensionales compuestos por células que pueden estar en uno de tres estados: reposo, excitado y refractario. Estos autómatas son particularmente útiles para simular patrones de propagación de la actividad eléctrica en tejidos cardiacos, así como otros fenómenos de propagación y ondas.

La evolución de las células en un autómata de Greenberg-Hastings se rige por reglas locales que determinan la activación y desactivación de las células en función de su estado actual y el estado de sus vecinos. Estas reglas son las siguientes:

- Reposo: una célula en estado de reposo se mantendrá en reposo a menos que al menos uno de sus vecinos esté en estado excitado. En ese caso, la célula pasará al estado excitado en el siguiente paso de tiempo.
- Excitado: una célula en estado excitado se moverá al estado refractario en el siguiente paso de tiempo, independientemente del estado de sus vecinos.
- Refractario: una célula en estado refractario se moverá al estado de reposo en el siguiente paso de tiempo, independientemente del estado de sus vecinos.

Estas reglas simples permiten la propagación de la excitación a través del autómata, creando patrones de propagación que pueden ser analizados y estudiados.

Los automátas celulares han sido una influencia en el mundo del videojuego. Existen diversos juegos y hasta géneros basados en autómatas celulares. El siguiente apartado tratará sobre los simuladores de arena y su relación con los autómatas celulares.

8.2 Simuladores de arena

En esta sección, se hablará de manera resumida acerca de qué son los simuladores de arena y su funcionamiento. Más tarde, se presentan una serie de antecedentes que se han tomado de base para el desarrollo del proyecto.

8.2.1 Introducción

Los simuladores de arena son simuladores cuyo objetivo es representar con precisión interacciones dinámicas entre elementos físicos como granos de arena u otros materiales granulares presentes en el mundo real. Estos comparten muchas similitudes estructurales y conceptuales con los autómatas celulares.

El «mapa» de la simulación está formado por un conjunto de celdas dentro de un número finito de dimensiones, representado como una matriz. 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. Cada uno de estos estados representa el tipo de la partícula que se encuentra en la celda.

Sin embargo, a diferencia de los autómatas celulares, donde la evolución de cada celda está estrictamente determinada por reglas locales con sus celdas vecinas y pueden procesarse todas las partículas sin seguir un orden específico, un simulador de arena funciona de manera secuencial y no determinista. El estado futuro de una celda no solo es influenciado por el estado de sus celdas vecinas, sino también por el orden en que las partículas son procesadas. Además, el procesar una celda, no necesariamente se limita a afectar solo a esa celda. Por ejemplo, el simular fenómenos físicos como la gravedad, implica mover partículas por la matriz de celdas, fenómeno no contemplado en los autómatas celulares. En este sistema, «mover» una partícula consiste en cambiar el estado de una celda a otra.

Sin embargo, existen casos en los que este comportamiento no es deseado. Es posible simular el comportamiento de un autómata celular en un simulador de arena aún procesando de forma secuencial mediante una técnica conocida como «doble buffer». En esta técnica, se tienen dos matrices, una que representa el estado actual de la simulación y otra que representa el estado futuro. En cada paso de la simulación, se procesa el estado actual y se guarda el resultado en el estado futuro. Una vez se ha procesado toda la matriz, se intercambian los estados de las matrices. De esta forma, se consigue que el estado futuro de una celda no se vea afectado por el estado futuro de otra celda, es decir, que una celda cambie su estado o el de su vecina no afecta a dicha celda en el mismo paso de simulación.

Traté de explicar de forma sencilla y a alta nivel sin tener que entrar en detalles de donde se lee y donde se escribe en cada una de las matrices, aún así... No estoy seguro de si está bien explicado o debería explayarme más. El párrafo de arriba es el resultado de pensar durante un buen rato como sintetizar mi conocimiento de forma clara y concisa. En cuanto a referencias, esta técnica suele usarse en GPU por razones distintas, los libros que tengo de autómatas celulares no mencionan nada similar así que no puedo referenciarlos.

En un simulador de arena, pueden existir multitud de tipos de partículas, cada una con unas reglas distintas de evolución e interacción con otras celdas de la matriz, lo que puede dar lugar a ejecuciones con dinámicas muy complejas. Para explicar el funcionamiento de los simuladores

de arena, se tomará un ejemplo básico de un simulador que contenga solo un tipo de partícula, la de arena. Esta partícula tiene el siguiente comportamiento:

- Si la celda de abajo esta vacía, me muevo a ella.
- En caso de que la celda de abajo esté ocupada por otra partícula, intento moverme en dirección abajo izquierda y abajo derecha, si las celdas están vacías.
- En caso de que no se cumpla ninguna de estas condiciones, la partícula no se mueve.

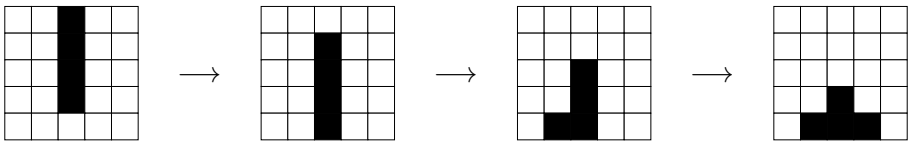


Figura 10: Ejemplo de simulador de arena

Esta ejecución toma como base un orden de ejecución de abajo a arriba y de derecha a izquierda. Para un orden de ejecución de arriba a abajo y de izquierda a derecha el resultado sería el siguiente.

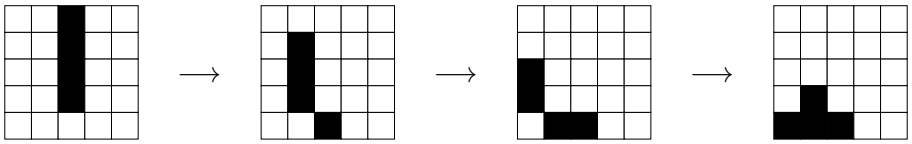


Figura 11: Ejemplo de simulador de arena, diferente orden

Como puede verse, el orden en el que se procesan las celdas de la matriz afecta al resultado final.

8.2.2 Simuladores de arena dentro de los videojuegos

Dentro de la industria de los videojuegos, se han utilizado simuladores de arena con diferentes fines, como pueden ser 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 siguiendo unas reglas físicas y químicas específicas de ese material. Esto permite que los diferentes materiales sólidos, líquidos y gaseosos se comporten de manera realista de acuerdo a sus propiedades. El jugador tiene la capacidad de provocar reacciones en este entorno, por ejemplo destruyéndolo o haciendo que interactúen entre sí.

Figura 12: Imagen gameplay de Noita

Noita no es el primer videojuego que hace uso de los simuladores de partículas. A continuación, se enumeran algunos de los títulos, tanto videojuegos como sandbox, más notables de simuladores de los cuales el proyecto ha tomado inspiración durante el desarrollo.

- Falling Sand Game [12]

Probablemente el primer videojuego comercial de este 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 estableció una base que luego tomaron otros videojuegos más adelante.

- Powder Toy [13]

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 [14]

Este proyecto utiliza la misma base que sus predecesores, 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 más limitada que la de proyectos anteriores. De Sandspiel, nace otro proyecto llamado Sandspiel Club [15], 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 [16] 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.

Proximo capítulo

Cuanto más grande sea el tamaño de la matriz que represente el estado de un autómata celular, menor será el rendimiento, ya que procesar las reglas de evolución de cada una de las celdas de manera secuencial, que es la forma que tiene la CPU de ejecutar instrucciones, llega a resultar muy costoso a medida que aumenta el número de celdas. Sin embargo, se puede aprovechar una particularidad de los autómatas celulares, y es que el cálculo de evolución de cada celda es independiente del cálculo de evolución del resto de celdas. Este tipo de problemas se conoce comúnmente como “Embarrassingly parallel”, problemas donde la sub-separación de una tarea grande en tareas muy pequeñas independientes entre ellas es obvia o directa.

Esto no ocurre de manera tan directa en los simuladores de arena, donde el orden de ejecución de las celdas afecta al resultado final, aunque se verá en el Sección 12 cómo modificando un poco las reglas, se puede conseguir transformar un problema de simulación de partículas en un problema "Embarrassingly parallel".

Existe un componente de ordenador que, por su arquitectura, es muy eficaz en la resolución de este tipo de problemas. Este componente se llama GPU, y se hablará de él a continuación.

9 Programación paralela

En esta sección se hablará sobre qué es la programación paralela, su funcionamiento y usos tanto en CPU como en GPU.

La programación paralela es una técnica de programación que consiste en dividir un problema en tareas más pequeñas y ejecutarlas simultáneamente en múltiples procesadores o núcleos de procesamiento. Esto permite aprovechar el poder de cómputo del hardware y acelerar la ejecución de programas.

Sin embargo, debido a las particularidades de cada tipo de hardware, la forma en que funciona y se aplica varía en función de si se quiere usar la CPU o la GPU.

Cabe destacar que independientemente del hardware utilizado, la paralelización de un problema no incrementa el rendimiento de manera lineal con el número de núcleos utilizados, el incremento de rendimiento tiene un límite.

La ley de Amdahl [17] es un principio importante en la programación paralela que establece que el rendimiento máximo que se puede lograr al paralelizar un programa está limitado por la fracción secuencial del código. En otras palabras, aunque se pueda paralelizar una parte del código, siempre habrá una porción que debe ejecutarse de forma secuencial y que limitará el rendimiento general del programa.

La ley de Amdahl se expresa mediante la siguiente fórmula:

$$\text{Mejora Total} = \frac{1}{(1-P) + \left(\frac{P}{N}\right)}$$

Donde:

- `Mejora Total` es la mejora en el rendimiento del programa al paralelizarlo.
- `P` es la fracción del código que se puede paralelizar.
- `N` es el número de núcleos de procesamiento (hilos) disponibles.

Esta fórmula muestra que, a medida que aumenta el número de procesadores o hilos (N), el rendimiento total mejora, pero solo hasta cierto punto. La fracción secuencial del código (1 - P) siempre limitará el rendimiento máximo que se puede lograr.

La intro que he hecho es mejorable, pero es un apaño temporal para no tener directamente un subapartado sin nada de texto. Como ahora este capítulo es de programación paralela y no solo de GPU considero mejor hacerlo así. Podría hacerse un capítulo separado para GPU y CPU pero dado que de CPU se hablará menos creo que es mejor dejarlo aquí complementando al de GPU como se sugirió en las notas. Además de que no he revisado el apartado de GPU, estoy escribiendo mi parte para poder hacer los demás apartados, seguramente nos falte ponernos de acuerdo en este apartado y redactarlo mejor entre los dos, pero en principio la información que tenemos debería ser la adecuada y solo falta presentarla de forma correcta. Por cierto, este mensaje es para Jonathan, esto debería haberse resuelto antes de enviarse a Pedro Pablo

9.1 Programación paralela en GPU

La GPU (graphics processing unit) es un procesador originalmente diseñado para manejar y acelerar el procesamiento de tareas gráficas, como puede ser el mostrar imágenes o vídeos en pantalla. Para facilitar la aceleración de estas tareas, se crearon los shaders, pequeños programas

gráficos destinados a ejecutarse en la GPU como parte del pipeline gráfico. El pipeline gráfico es el conjunto de operaciones secuenciales que finalmente formarán la imagen a mostrar en pantalla. La denominación pipeline hace referencia a que las operaciones que lo componen se ejecutan de manera secuencial y cada operación recibe una entrada de la fase anterior y devuelve una salida que recibirá la siguiente fase como entrada, hasta completar la imagen. [18]

El pipeline gráfico comienza con la representación de objetos mediante vértices. Cada vértice contiene información como su posición, normal, color y coordenadas de textura.

Luego, las coordenadas de los vértices se transforman en coordenadas normalizadas mediante matrices de transformación, pasando por etapas de escena, vista y proyección.

Después, se ensamblan los vértices para formar primitivas, se descartan o recortan las que están fuera del campo visual y se mapean a coordenadas de pantalla.

La rasterización determina qué píxeles formarán parte de la imagen final, utilizando un buffer de profundidad para determinar qué fragmentos se dibujan. Se interpola entre los atributos de los vértices para determinar los atributos de cada fragmento y se decide el color de cada píxel, considerando la iluminación, la textura y la transparencia.

Finalmente, los fragmentos dibujados se muestran en la pantalla del dispositivo.

Aunque la funcionalidad inicial de la GPU se limitaba al apartado gráfico, los fabricantes de este tipo de chips se dieron cuenta de que los desarrolladores buscaban formas de mapear datos no relacionados con imágenes a texturas, para así ejecutar operaciones sobre estos datos mediante shaders [19]. Esto significaba que se podía aprovechar las características de este hardware para la resolución de tareas que no tengan que ver con imágenes, por lo que extendieron su uso más allá de la generación de gráficos.

Una de estas extensiones fue la creación de «compute shaders» [20] que son programas diseñados para ejecutarse en la GPU, pero, a diferencia de los shaders, no están directamente relacionados con el proceso de renderizado de imágenes, por lo que se ejecutan fuera del pipeline gráfico. Los «compute shaders» se emplean para realizar cálculos destinados a propósitos que se benefician de la ejecución masivamente paralela ofrecida por la GPU. Son ideales para tareas como simulaciones físicas, procesamiento de datos masivos o aprendizaje automático.

9.1.1 Arquitectura GPU

Para lograr el procesamiento de shaders de la manera más eficiente, la GPU se diseñó con una arquitectura hardware y software que permite la paralelización de cálculos en el procesamiento de vértices y píxeles independientes entre sí.

Este apartado se centra en explicar las diferencias de arquitectura entre una CPU y una GPU a nivel de hardware, así como en explicar cómo este hardware interactúa con el software destinado a la programación de GPUs.

9.1.1.1 Hardware

La tarea de renderizado requería de un hardware diferente al presente en la CPU debido a la gran cantidad de cálculos matemáticos que requiere. Desde transformaciones geométricas hasta el cálculo de la iluminación y la aplicación de texturas, todas estas tareas se basan en

manipulaciones matemáticas haciendo uso de vectores y matrices. Para optimizar el proceso de renderizado, es esencial reducir el tiempo necesario para llevar a cabo estas operaciones [21].

Por lo tanto, surge la GPU como co-procesador con una arquitectura SIMD (single instruction multiple data) cuya función es la de facilitar a la CPU el procesado de tareas relacionadas con lo gráfico, como renderizar imágenes, vídeos, animaciones, etc [17].

Al ser el objetivo de la GPU el procesar tareas de manera paralela, se puede observar una gran diferencia en cuanto a la distribución de espacio físico (recuento de transistores) dentro del chip con respecto a la CPU, que esta diseñada para procesar las instrucciones secuencialmente [1].

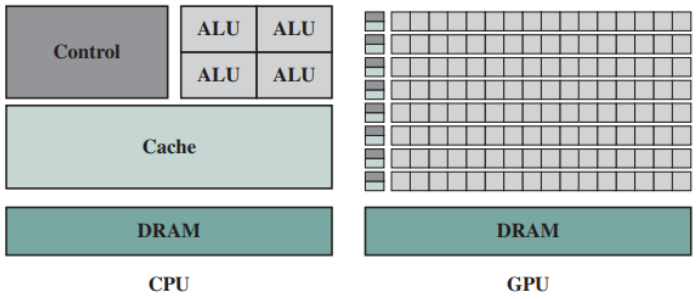


Figura 13: Comparativa arquitectura de un chip de CPU y de GPU [1]

Una GPU dedica la mayor cantidad de espacio a alojar núcleos para tener la mayor capacidad de paralelización posible, mientras que la CPU dedica, la mayoria de su espacio en chip a diferentes niveles de caché y circuitos dedicados a la logica de control [1].

La CPU necesita estos niveles de caché para intentar minimizar al máximo los accesos a memoria principal, los cuales ralentizan mucho la ejecución. De igual manera, al estar diseñados los núcleos CPU para ser capaces de ejecutar cualquier tipo de instruccion, requieren lógica de control para gestionar los flujos de datos, controlar el flujo de instrucciones, entre otras funciones.

Sin embargo, la GPU al estar dedicada principalmente a operaciones matemáticas y por lo tanto tener un set de instrucciones mucho más reducido en comparación con la CPU, puede prescindir de dedicarle espacio a la logica de control. Al acceder a memoria, a pesar de que los cores tengan registros para guardar datos, la capacidad de estos es muy limitada, por lo que es común que se acceda a la VRAM (Video RAM). La GPU consigue camuflar los tiempos de latencia manejando la ejecucion de hilos sobre los datos. Cuando un hilo esta realizando acceso a datos, otro hilo está ejecutandose [1].

La gran cantidad de cores presentes en una GPU, están agrupados en estructuras de hardware llamados SM (Streaming Multiprocessors) en NVIDIA y CP (Compute Units) en AMD. NVIDIA y AMD son dos de los principales fabricantes de tarjetas gráficas, cada uno con su propia arquitectura y tecnologías específicas. Además de núcleos de procesamiento, estas agrupaciones incluyen normalmente una jerarquía básica de memoria con una cache L1, una memoria compartida entre núcleos, una caché de texturas, un programador de tareas y registros para almacenar datos. Su tarea principal es ejecutar programas SIMT (single-instruction multiple-thread) correspondientes a un kernel, asi como manejar los hilos de ejecución, liberándolos una vez que han terminado y ejecutando nuevos hilos para reemplazar los finalizados. [21]

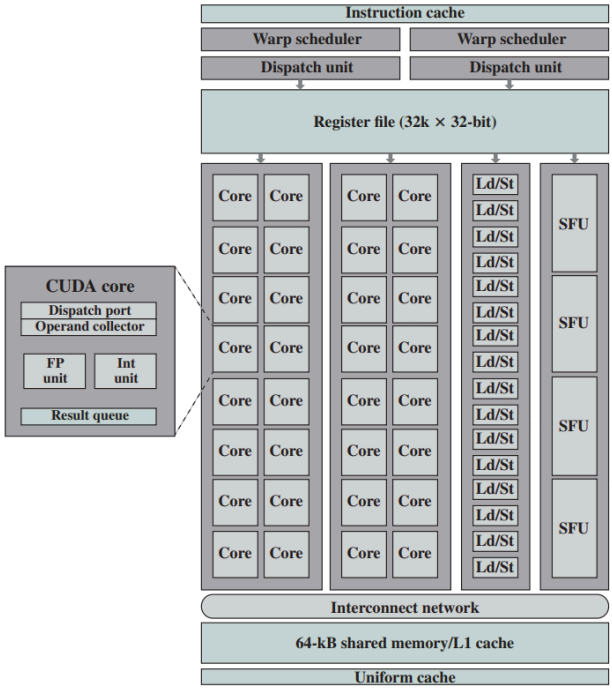


Figura 14: Streaming Multiprocessor [1]

9.1.1.2 Software

Debido a que la implementacion de CUDA fue un punto de inflexión en el desarrollo de GPUs y asentó las bases de lo que hoy es la computación de propósito general en unidades de procesamiento gráfico, se explicará cómo se enlaza el software al hardware ya explicado haciendo uso de CUDA. Todos los conceptos son extrapolables a otras APIs de desarrollo como pueden ser SYCL o OpenMP.

Un programa CUDA puede ser dividido en 3 secciones [1]:

- Código destino a procesarse en el Host (CPU).
- Código destinado a ser procesado en el Dispositivo (GPU).
- Código que maneja la transferencia de datos entre el Host y el dispositivo.

El código destinado a ser procesado por la GPU se conoce como kernel. Un kernel está diseñado para contener la menor cantidad de código condicional posible. Esto se debe a que la GPU está optimizada para ejecutar un mismo conjunto de instrucciones en múltiples datos de manera simultánea. Cuando hay muchas ramificaciones condicionales (como if-else), puede haber una divergencia en la ejecución de los hilos, lo que disminuye la eficiencia del paralelismo y puede resultar en un rendimiento inferior.

A cada instancia de ejecución del kernel se le conoce como hilo. El desarrollador define cual es el número de hilos sobre los que quiere ejecutar el kernel, idealmente maximizando la paralelización de los cálculos. Estos hilos pueden ser agrupados uniformemente en bloques, y a su vez estos bloques son agrupados en un grid de bloques. El número de bloques totales que se crean viene dictado por el volumen de datos a procesar. Tanto los bloques como los grids pueden tener de 1 a 3 dimensiones, y no necesariamente tienen que coincidir.

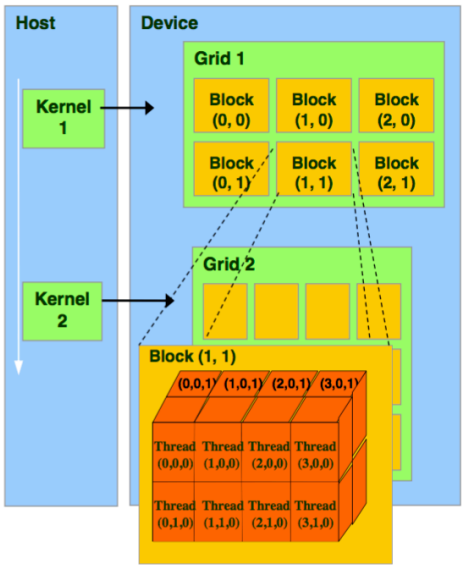


Figura 15: Jerarquía de ejecución [2]

El «scheduler global» en una GPU tiene la función principal de coordinar y programar las unidades de procesamiento disponibles para ejecutar tareas en paralelo. A la hora de ejecutar el kernel, este scheduler crea warps, sub-bloques de un cierto número de hilos consecutivos sobre los que se llevara a cabo la ejecución, que luego serán programados para ejecutar por el scheduler de cada SM. Es totalmente imprescindible que estos bloques de hilos puedan ser ejecutados 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 mediante esperas, mediante 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 dentro de un bloque necesitan residir en el mismo SM para poder compartir recursos de memoria.

9.2 Programación paralela en CPU

La CPU (central processing unit) es el procesador principal de un ordenador y se encarga de ejecutar las instrucciones de los programas. A diferencia de la GPU, la CPU está diseñada para ejecutar instrucciones secuencialmente, es decir, una instrucción tras otra. Sin embargo, la CPU también puede ejecutar tareas en paralelo, pero de una manera diferente a la GPU.

La programación paralela en CPU se basa en la creación de hilos de ejecución, que son unidades de procesamiento independientes que pueden ejecutar tareas simultáneamente. Debido a que los hilos pueden compartir memoria, es posible tanto resolver varios problemas a la vez como dividir un problema en tareas más pequeñas y ejecutarlas en paralelo.

Este acceso compartido a memoria incurre en una serie de problemas que se deben tener en cuenta a la hora de programar en paralelo en CPU. Los problemas mostrados a continuación no siguen un orden de importancia, ya que todos ellos son igual de importantes: Condiciones de carrera, interbloqueos y problemas de inanición.

Las condiciones de carrera se dan cuando dos o más hilos intentan acceder y modificar el mismo recurso al mismo tiempo. Esto puede llevar a resultados inesperados y errores en el programa.

Para evitar condiciones de carrera, se desarrollaron lo que se conoce como mecanismos de sincronización, como los semáforos o los mutex.

Un mutex es un recurso compartido entre hilos que permite controlar el acceso a una sección crítica del código. Cuando un hilo adquiere un mutex, ningún otro hilo puede acceder a la sección crítica (usualmente un recurso compartido) hasta que el primer hilo libere el mutex. Esto evita que se produzcan condiciones de carrera y garantiza que solo un hilo pueda acceder a la sección crítica en un momento dado.

Los interbloqueos y problemas de inanición son problemas resultantes del uso incorrecto de los mecanismos de sincronización. Un interbloqueo ocurre cuando dos o más hilos se bloquean mutuamente, es decir, cada hilo espera a que otro hilo libere un recurso que necesita, lo que resulta en que ninguno de los hilos pueda avanzar. La inanición, por otro lado, ocurre cuando un hilo es incapaz de avanzar debido a que otros hilos tienen prioridad sobre él, lo que resulta en que el hilo «hambriento» no pueda completar su tarea al no recibir acceso a los recursos necesarios.

Existe una alternativa para evitar estos problemas relacionados con los mecanismos de sincronización: `canales`. Los canales son estructuras de datos que permiten la comunicación entre hilos de manera segura y eficiente. Cada hilo puede enviar y recibir mensajes a través de un canal, lo que evita la necesidad de utilizar mecanismos de sincronización y reduce la posibilidad de condiciones de carrera.

El uso de canales posibilita un tipo de balance de trabajo entre hilos llamado `work stealing` [22]. En este modelo, los hilos ejecutan tareas, al terminar, envía un mensaje a otro hilo para que le envíe una tarea. De esta forma, los hilos que terminan antes pueden ayudar a los que todavía tienen tareas pendientes, evitando la inanición y mejorando el rendimiento del programa.

No estoy seguro de si he explicado el work stealing de la forma más correcta, pero es la forma en la que se ha implementado en nuestro caso. Cuando programé el multithread en Lua ni siquiera sabía que estaba usando un patrón llamando work stealing, me di cuenta más tarde al investigar sobre el tema. En mi caso mi motivación para implementar este sistema no es la inanición, era que crear 8-16 threads 60 veces por segundo era costoso, por lo que esta era la form más sencilla de crear los threads una vez y reutilizarlos. No estoy muy seguro de esta última parte, quizás pueda mencionarlo como una técnica similar al work stealing que no es «puramente» work stealing. Y es algo particular de nuestra implementación, no he visto que implementarlo de esta forma sea algo común o al menos algo que esté documentado. De todas formas voy a dejarlo aquí en estado del arte porque sino sé que luego habrá una nota diciendo que ya debería haber sido explicado antes… Y no me gusta tener que explicarlo aquí porque luego al explicar el simulador solo referencio esta parte y ale, cuadno este sistema es una parte importante de nuestra contribución y ponerlo aquí me parece que lo desmerece.

No estoy seguro de si debería añadir algo más aquí.No he querido ir a cosas más técnicas como que problemas de sincronización de caché y demás ya que los lenguajes de alto nivel se encargan de estas cosas y no es algo que nos afectara en el desarrollo del proyecto (a diferencia de los 3 problemas expuestos que sí los padecimos). En este apartado solo he mencionado las cosas relevantes de procesamiento paralelo en CPU que nos afectaron en el desarrollo del proyecto. (por eso menciono los canales y el work stealing)

10 Estrategias para definir comportamiento en motores de videojuegos

A la hora de crear un videojuego, existen varias estrategias a la hora de diseñar la arquitectura de software. Existe la posibilidad de programar el comportamiento del juego de forma directa, sin embargo esto resulta en un sistema difícil de modificar y reutilizar. Debido a esto se han creado motores de videojuegos. Un motor de videojueogs [23] es un conjunto de herramientas que permiten a los desarrolladores crear videojuegos de forma más sencilla. Un motor de videojuegos puede o no tener una interfaz visual. Dicho conjunto de herramientas permite crear el «gameplay» o comportamiento del juego.

Un motor de videojuegos puede proveer más o menos elementos reutilizables. Sin embargo, mientras más elementos o comportamientos predefinidos tenga el motor, más especializado será, lo cual límita su capcidad de extensibilidad. La forma en que se desarrollan los comportamientos del juego con el motor depende de este mismo. Existen diferentes opciones, cada una con sus ventajas y desventajas.

En este capítulo se van a explorar tres posibilidades para desarrollar el sistema de «scripting» de un motor de videojuegos. Estas son: ficheros de definición de datos, librerías dinámicas y lenguajes de scripting.

10.1 Ficheros de definición de datos

Una opción es usar ficheros de definición de datos, archivos que como su nombre indican, solo contienen datos. Suelen ser representados con un formato legible (YAML, JSON…) puesto que están pensados para ser editados por humanos que además, pueden no ser programadores. Al implementar esta opción, los juegos resultantes son «dirigidos por datos» [24], debido a que en lugar de especificar el comportamiento del juego mediante código, se configura mediante estos ficheros. Esto permite separar el trabajo de los programadores de los diseñadores de niveles o de juego.

Un ejemplo sencillo de esto es un juego bullet hell. En lugar de especificar mediante código cada patrón, velocidad, tiempos, etc. Se puede definir un fichero que el motor carga durante la ejecución del juego y contiene todos los parámetros necesarios para crear las balas. A continuación muestra un ejemplo de un posible fichero de definición de datos para un juego bullet hell.

```yaml
define_action:
  name: "shoot"
  behaviour:
    type: "linear"
    speed: 5
    direction: 90
    time: 1

define_action:
  name: "spawnLots"
  behaviour:
    type: "spawn"
    amount: 10
    do_action:
```

```
15        type: "shoot"
16        direccion: random 0 360
17
18  level:
19    do_action: "spawnLots"
20    wait: 5
21    repeat 10:
22      do_action: "spawnLots"
23      wait: 0.5
```

Debido a que el motor debe contener el código necesario para poder interpretar estos ficheros, existe una gran dependencie con el motor y las opciones de extensibilidad sean limitadas. En este tipo de sistemas no es posible crear juegos radicalmente distintos con solo cambiar los datos debido a estas limitaciones.

10.2 Librerías dinámicas

Para superar las restricciones del modelo anterior, es necesario que el motor sea más flexible, es decir, debe poder permitir definir comportamiento permitiendo mayor libertad. Para ello, se pueden usar librerías dinámicas.

Una librería dinámica [25] es un archivo que contiene código compilado que puede ser cargado y vinculado a un programa en tiempo de ejecución. Esto significa que el programa no necesita incluir este código en su propio archivo binario, sino que puede cargarlo cuando se necesita. Esto permite definir comportamiento con mucha más flexibilidad. Siguen existiendo limitaciones por parte del motor, pues el desarrollador solo puede usar las funciones del motor que exponga mediante su API. A pesar de ello, esta alternativa resulta ser mucho más versátil que la anterior.

El uso de librerías dinámicas no es exclusivo de los motores de videojuegos. Muchos programas de software utilizan este mecanismo para extender su funcionalidad.

Las librería dinámicas no son un formato universal, sino que cada sistema operativo tiene su propio mecanismo para cargar y vincular librerías dinámicas. Por ejemplo, en Windows se utilizan archivos DLL, en Linux se utilizan archivos SO y en macOS se utilizan archivos dylib. Esto significa que las librerías dinámicas no son necesariamente portables entre sistemas operativos, lo que puede ser una limitación en algunos casos. No obstante, existe un consenso al respecto a su funcionalidad. Por ejemplo, los grandes sistemas operativos actuales, Windows, Linux y MacOS, permiten definir callbacks o funciones para que una librería detecte cuando se ha cargado o descargado por el programa principal. Usar una dependencia que solo funcione en un sistema operativo podría dificultar la portabilidad del juego. Para mantener la portabilidad del sistema es importante ser cuidadodoso con las dependencias que se usan.

A pesar de su versatilidad, el uso de librerías dinámicas tiene ciertos problemas. Como se ha dicho, una librería dinamica contiene código compilado, esto implica que durante el desarrollo del juego, cada vez que se modifique el código de la librería, es necesario recompilarla y recargarla en el motor. Esto puede resultar en un proceso tedioso y lento que afecte de forma significativa al flujo de trabajo. Para evitar este problema, se pueden usar lenguajes de scripting.

10.3 Lenguajes de scripting

Un lenguaje de scripting [26] es un lenguaje de programación que se utiliza para controlar la ejecución de un programa o para extender su funcionalidad. A diferencia de los lenguajes

de programación tradicionales, los lenguajes de scripting suelen ser interpretados en lugar de compilados.

Un lenguaje de programación compilado es aquel que se transforma en código máquina mediante un proceso conocido como compilación, antes de su ejecución. Durante la ejecución, el sistema operativo carga este código máquina en la memoria y lo ejecuta directamente.

Por otro lado, un lenguaje de programación interpretado no se traduce previamente a código máquina. En su lugar, se traduce y ejecuta simultáneamente durante el tiempo de ejecución por un programa llamado intérprete. Este proceso de traducción incurre en un sobrecoste que hace que los lenguajes interpretados sean más lentos que los compilados.

Sin embargo, los lenguajes de scripting suelen ser más fáciles de aprender y de utilizar que los lenguajes de programación tradicionales. Esto se debe a que los lenguajes de scripting suelen tener una sintaxis más sencilla y menos reglas que los lenguajes de programación compilados. Suelen ser lenguajes que gestionan la memoria automáticamente, es decir, no es necesario liberar la memoria que se ha reservado, lo que facilita la programación.

Uno de los lenguajes de scripting más usados para modificar e incluso desarrollar videojuegos es Lua [27]. Lua es un lenguaje de scripting de alto nivel, multi-paradigma, ligero y eficiente, diseñado principalmente para la incorporación en aplicaciones. Fue creado en 1993 por Roberto Ierusalimschy, Luiz Henrique de Figueiredo y Waldemar Celes, miembros del Grupo de Tecnología en Computación Gráfica (Tecgraf) de la Pontificia Universidad Católica de Río de Janeiro, Brasil.

Lua es conocido por su simplicidad, eficiencia y flexibilidad. Su diseño se centra en la economía de recursos, tanto en términos de memoria como de velocidad de ejecución. Existe una única estructura de datos, la tabla, que se utiliza para representar tanto arrays como diccionarios. Además, para poder «heredar» funciones de una tabla, Lua define el concepto de metatabla. En Lua, cada tabla puede tener asociada una metatabla. Cuando el desarrollador llama a una función y esta no está definida en la tabla, Lua busca en la metatabla de la tabla para ver si la función está definida ahí. Esto es comparable a los prototipos en JavaScript.

Una de las características más destacadas de Lua es su capacidad para ser embebido en aplicaciones [28]. Esto se debe a su diseño como un lenguaje de scripting, que permite que el código Lua sea llamado desde un programa en C, C++ u otros lenguajes de programación. Esta característica ha llevado a que Lua sea ampliamente utilizado en la industria de los videojuegos, donde se utiliza para controlar la lógica del juego y las interacciones del usuario. Existen motores como Defold que integran Lua como lenguaje de scripting o el popular juego Roblox, que permite a los usuarios crear sus propios juegos utilizando una version modificada de Lua llamada Luau.

Sin embargo, esta flexibilidad es un arma de doble filo. Lua puede ser integrado en aplicacions escritas en diversos lenguajes. A menudo será necesario enviar y recibir datos entre el programa principal y el código de Lua. Esta comunicación se realiza mediante un mecanismo llamado FFI o «Foreign Function Interface» (interfaz de funciones foráneas) por sus siglas en inglés. Esta barrera entre ambos lenguajes provoca que la comunicación entre ellos sea más lenta que si se usara un solo lenguaje de programación. Para cada función del API del sistema en el lenguaje principal hay que crear un «binding» o nexo con el lenguaje de scripting. Esto se denomina

«glue code» o código de pegamento. Si en algún momento el sistema cambia y por tanto su API, habrá que ajustar el código pegamento para que siga funcionando.

Como se ha mencionado, los lenguajes de scripting son interpretados. Sin embargo, existe una técnica llamada compilación Just-In-Time (JIT) que permite compilar el código de un lenguaje de scripting en código máquina durante la ejecución. Esto permite que el código sea ejecutado de forma más rápida, ya que no es necesario interpretarlo en tiempo real. LuaJIT [28] es una implementación de Lua que utiliza esta técnica y que ha demostrado ser muy eficiente en términos de velocidad de ejecución.

Sin embargo, cabe destacar que en los lenguajes JIT, puede llegar a ser importante saber ciertos detalles de su funcionamiento interno para poder aprovechar su potencial. Como se mencionó anteriormente, la única estructura de datos en Lua es la tabla, no existen tipos salvo los primitivos (números, booleanos y cadenas de texto). Esto significa que algunas optimizaciones dependen del uso que el desarrollador haga del lenguaje. Por ejemplo, si una función recibe dos parámetros y esos dos parámetros siempre son números, LuaJIT puede optimizar la función en base a heurísticas [29], [28]. Sin embargo, si durante la ejecución se llama a la función con otro tipo de dato, LuaJIT desactivará la optimización y la función se ejecutará de forma más lenta. Este tipo de optimizaciones son comunes en los lenguajes JIT, por lo que es importante tener en cuenta cómo funcionan para poder aprovechar su potencial.

10.4 Blockly

Este apartado tiene como objetivo explicar qué es Blockly así como su funcionamiento.

Blockly es una biblioteca perteneciente a Google lanzada en 2012 que permite a los desarrolladores la generación automática de código en diferentes lenguajes de programación mediante la creación de bloques personalizados. Fue diseñado inicialmente para generar código en JavaScript, pero debido a su creciente demanda, se ha adaptado para admitir de manera nativa la generación de código en una amplia variedad de lenguajes de programación: JavaScript, Python, PHP, Lua y Dart.

Esta biblioteca es cada vez más conocida y usada en grandes proyectos. Actualmente se emplea en algunos como 'App Inventor' [30] del MIT [31], para crear aplicaciones para Android, 'Blockly Games' [32], que es un conjunto de juegos educativos para enseñar conceptos de programación que también pertenece a Google, 'Scratch' [33], web que permite a jóvenes crear historias digitales, juegos o animaciones o Code.org [34] para enseñar conceptos de programación básicos.

Blockly opera del lado del cliente y ofrece un editor dentro de la aplicación, permitiendo a los usuarios intercalar bloques que representan instrucciones de programación. Estos bloques se traducen directamente en código según las especificaciones del desarrollador. Para los usuarios, el proceso es simplemente arrastrar y soltar bloques, mientras que Blockly se encarga de generar el código correspondiente. Luego, la aplicación puede ejecutar acciones utilizando el código generado.

A continuación, se detalla tanto la interfaz que se muestra al usuario de la aplicación como el funcionamiento y proceso de creación de bloques y de código a partir de ellos.

Un proyecto de Blockly se tiene la siguiente estructura [35]:

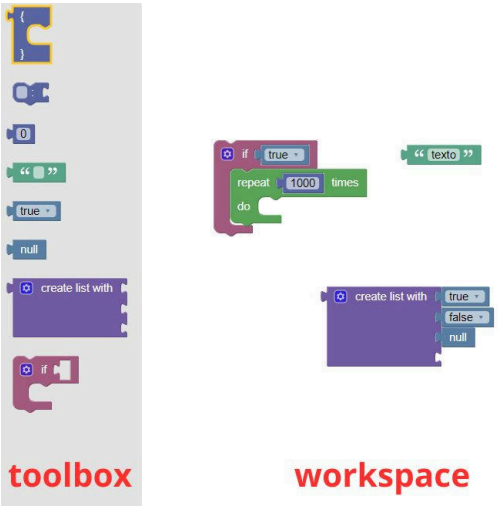


Figura 16: estructura básica de Blockly

La figura Figura 16 muestra la estructura de un proyecto de Blockly. Esta se divide en 2 partes básicas, la toolbox o caja de herramientas y el workspace o espacio de trabajo. La toolbox alberga todos los bloques que haya creado el desarrollador o que haya incluido por defecto. Estos bloques los puede arrastrar al workspace para generar lógica que haya sido definida por el desarrollador. Por defecto, todos los bloques son instanciables las veces que sean necesarias.

Para que el usuario pueda usar un bloque correctamente, son necesarios tres pasos desde el punto de vista del desarrollador [36]:

- Definir cómo es su apariencia visual. Esto puede ser realizado tanto usando código JavaScript como mediante JSON. Es recomendable usar JSON, aunque por características particulares puede ser necesario definirlos mediante JavaScript.
- Especificar el código que será generado una vez haya sido arrastrado el bloque al workspace. Debe haber una definición del código a generar por cada bloque y lenguaje que se quiera soportar.
- Incluirlo en la toolbox para que pueda ser utilizado. Esto puede ser realizado mediante XML O JSON, aunque Google recomienda el uso de JSON.

La definición de la apariencia y la inclusión en la toolbox son tareas bastante directas. Sin embargo, la generacion de código requiere la presencia de un intérprete que genere código a partir del texto que devuelve la función. En caso de querer generar código para un lenguaje no soportado por defecto, el desarrollador necesitará crear este intérprete.

En el caso de este proyecto, por optimización se decidió que la definición de partículas se diera a traves de JSON, y que ese JSON se parseara a código Rust. Debido a que JSON no requiere especificar la ejecución de nada, solo definir el texto de una manera correcta, esta implementación no resultó compleja.

11 Simuladores de arena en CPU

Este trabajo trata sobre simuladores de arena y, como se mencionó en la Sección 8.2, los simuladores de arena no son paralelizables debido a que cada celda puede modificar el estado de las demás. En este capítulo se muestran distintas implementaciones de simuladores de arena que se ejecutan en la CPU para poder compararlos.

Para poder realizar la comparativa, se han realizado 3 simuladores diferentes basados en explotar la CPU. Cada uno de ellos tiene sus propias ventajas y desventajas, además de distintos propósitos.

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

11.1 Generalidades

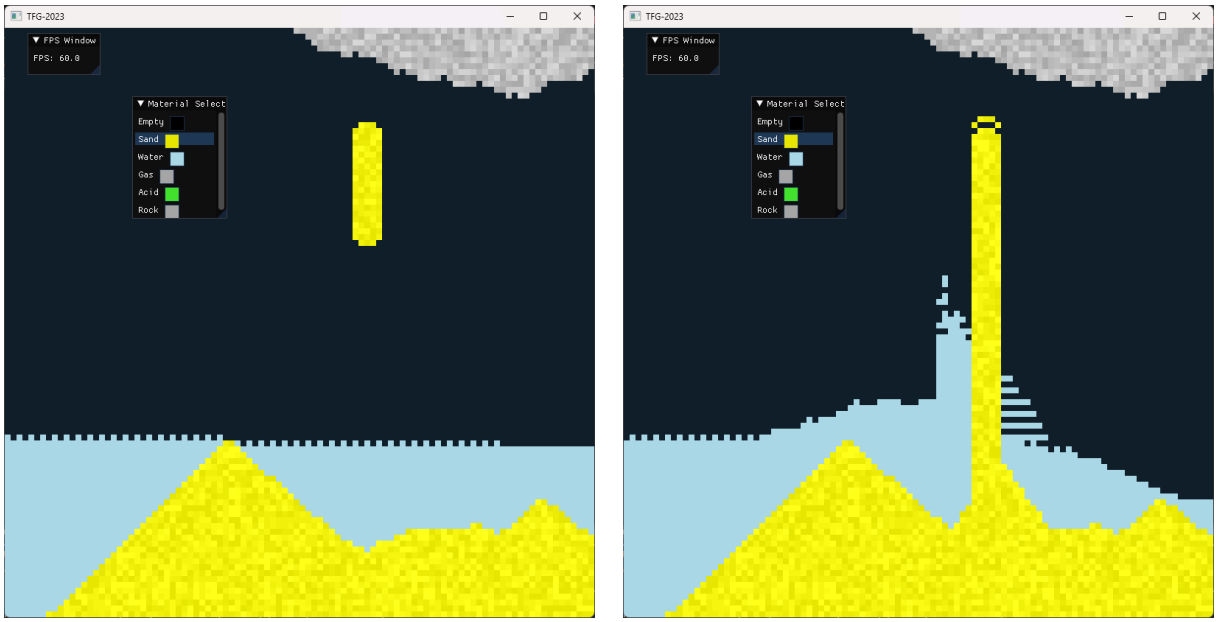
En todas las implementaciones, una partícula es una estructura de datos con al menos dos propiedades: id y clock. La propiedad `id` es un valor que indica el tipo de partícula, mientras que `clock` es un valor que alterna entre 0 y 1 en cada iteración. Esto permite que una partícula no sea procesada dos veces en la misma generación. Es decir, si una partícula de arena se mueve «hacia abajo» y la actualización del simulador procesa las partículas de arriba a abajo, la partícula de arena que se movió hacia abajo volverá a ser procesada dentro de la misma generación. Para evitar este problema, se usa el valor `clock` para marcar si una partícula fue procesada en la generación actual. Para evitar tener que resetear el valor de `clock` de todas las partículas en cada generación, se alterna entre 0 y 1 en cada iteración y se compara con el valor clock del sistema, que también alterna entre 0 y 1 en cada iteración.

11.2 Simulador en C++

El primer simulador fue desarrollado en C++ con OpenGL y GLFW. Este simulador sirve como base comparativa de las siguientes implementaciones. Este sistema posee 6 partículas: Arena, Agua, Aire, Gas, Roca y Ácido. En este sistema las partículas están programadas en el sistema y no son modificables de forma externa. Cada partícula tiene una serie de propiedades: color, densidad, granularidad, id y movimiento. El color es el color de la partícula, la densidad es un valor númerico que indica la pesadez relativa respecto otras partículas, la granularidad es un valor que modifica ligeramente el color de la partícula, la id es un valor que indica el tipo de partícula y el movimiento es una serie de valores que describe el movimiento de la partícula. Estos rasgos son particulares de esta implementación y no se repiten en las siguientes a excepción del identificador de la particula, que es común a todas las implementaciones.

Este sistema es limitado, pues los comportamientos de las partículas dependen de estos parámetros y el sistema que las procesa. Con todo, esto permite generar variaciones de partículas con facilidad. El valor del movimiento permite crear los tipos de movimientos más comunes (arena, agua, lava, gas, etc). La densidad permite controlar que partícula puede intercambiarse por otra sin tener que controlarlo manualmente para cada partícula… Al ser un sistema cerrado, se da lugar a un sistema más rápido y eficiente que los siguientes ya que el compilador puede optimizar el código de forma más eficiente.

La Figura 17 muestra la interacción entre partículas en este simulador en un mundo de $100 * 100$ celdas.



(a) Antes de que la arena llegue al agua (b) Arena hundiéndose en el agua

Figura 17: Interacción entre partículas en el simulador de C++

Esta implementación ejecuta una lógica directa en un solo hilo. Debido a esto es la base para comparar el rendimiento de las siguientes implementaciones.

Para poder mostrar el estado de la simulación de forma visual, se escribe en un buffer el color de cada partícula después de cada paso de simulación. Este buffer se envía a la GPU para ser renderizado en pantalla.

Esto lo cuento aquí porque en Lua y Rust lo hago distinto y no hay ningún libro o consenso al respecto de como hacerlo, es una decisión nuestra por lo cual no quería meterlo en estado del arte. En Lua es igual pero en multihilo, y en Rust el buffer se modifica al modificar una particula porque en esa simulación nos dimos cuenta de que actualizar el buffer cada vez que editas una partícula es más eficiente que hacerlo al final de la generación. ¿Es injusto de cara a la comparación? Quizás, pero que hacemos si no, ¿no lo contamos? Porque la versión de C++ no se ha tocado desde diciembre y por como está implementada sería un poco complicado porque la clase que ejecuta la lógica está totalmente separada de la clase que renderiza. La alternativa sería evitar hablar de este tema directamente o modificar la simulación en C++... Por ahora nos estamos centrando en pulir la memoria y si hay tiempo quizás podemos cambiar la versión de C++ y volver a ejecutar las pruebas de rendimiento.

Por otro lado, en las notas mencionas que no contamos suficiente en este apartado, pero es que no hay más, esta es una implementación simple, cerrada y eficiente que sirvió de base a las demás, pero más allá de que tiene una serie de parámetros que ya se han mencionado no tiene nada más destacable que contar, la chicha está en las otras implementaciones. (que en esas hay mucho más de lo que parece)

11.3 Simulador en Lua con LÖVE

LÖVE [37] es un framework de desarrollo de videojuegos en Lua orientado a juegos 2D. Permite dibujar gráficos en pantalla y gestionar la entrada del usuario sin tener que preocuparase de la

plataforma en la que se ejecuta. LÖVE usa LuaJIT, por lo que es posible alcanzar un rendimiento muy alto sin sacrificar flexibilidad.

Para mejorar aún más el rendimiento, esta implementación se basa de la librería FFI de LuaJit. Esta permite a Lua interactuar con código de C de forma nativa. Además, al poder declarar structs en C, es posible acceder a los datos de forma más rápida que con las tablas de Lua y consumir menos memoria. En este sistema, una particula es un struct en C que contiene su id y su clock.

Para facilitar la extensión y usabilidad de esta versión, se creó un API que permite definir partículas en Lua de forma externa. Una particula está definida por su nombre, su color y una función a ejecutar. Una vez hecho esto, el usuario solo debe arrastrar su archivo a la ventana de juego para cargar su «mod».

La función que ejecuta cada partícula recibe un solo parámetro: un objeto API. Este define las funciones necesarias para definir las reglas que modelan el comportamiento de una partícula. Además, para facilitar el desarrollo, las direcciones que consume el API son relativas a la posición de la partícula que se está procesando. Obtener el tipo de una partícula a la derecha de la actual es tan sencillo como llamar a `api:get(1, 0)`.

No sé si tenga sentido enumerar aquí las funciones que tiene dicho API… Tenemos un documento PDF con dichas funciones enumeradas, explicadas y hasta con ejemplos, por eso en la memoria anterior se menciona un «PDF anexo»

Además de esto, el sistema registra automáticamente los tipos de partículas en una tabla global que actúa como un enum. Esto permite que el usuario pueda comprobar con facilidad si un tipo de partícula está definido en el sistema para poder interactuar con este. Por ejemplo, una particula de lava puede comprobar si hay agua debajo de ella y convertirla en roca.

Sin embargo, simular tantas partículas es un proceso costoso. Debido a esto, se optimizó mediante la implementación de multihilo.

11.3.1 Multithreading en Lua

Si bien Lua es un lenguaje muy sencillo y ligero, tiene ciertas carencias, una de ellas es el multithreading. Lua no soporta multithreading de forma nativa. La alternativa a esto es instanciar una máquina virtual de Lua para cada hilo, esto es exactamente lo que love.threads hace. LÖVE permite crear hilos en Lua, pero además de esto, permite compartir datos entre hilos mediante `love.bytedata`, una tabla especial que puede ser enviada entre hilos por referencia, en resumen, un recurso compartido. Además de esto, LÖVE provee `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.

La implementación del multihilo dio lugar a problemas que no se tenían antes: escritura simultánea y condiciones de carrera. Esto supuso un desafío que fue resuelto implementando la actualización por bloques. En lugar de simular todas las particulas posible a la vez, se ejecutarían subregiones específicas de la simulación en 4 lotes. Se dividió la matriz en un «patron de ajedrez» [38]. Esto consiste en procesar primero las columnas y filas pares, luego columnas pares y filas impares… Y así hasta completar las 4 combinaciones posibles. Esto nos permite

dividir la actualización de la simulación en 4 «pases», donde en cada uno de estos pases se procesan varias partículas a la vez. La figura Figura 18 muestra como serían estos pases.

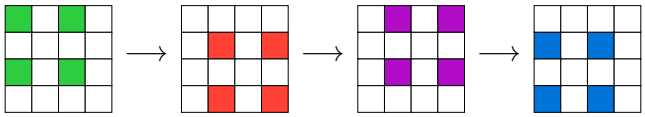


Figura 18: Patrón de ajedrez de actualización

Cada uno de los cuadrados de la imagen representa una «submatriz» de partículas, esto se denomina un «chunk». El sistema dividirá la matriz en chunks siguiendo el siguiente criterio: El tamaño mínimo de una chunk es de 16 píxeles, el número de chunks debe ser igual o superior al cuadrado de hilos disponibles. Estos requisitos garantizan que se puedan utilizar todos los hilos a la vez. El requisito de tamaño es para evitar que una partícula trate de modificar a otra lejana que esté siendo procesada por otro hilo.

El orden en que se procesan las regiones coincide con el mostrado en la Figura 18. Primero filas y columnas pares, a continuación, filas impares y columnas impares, tras esto, filas pares y columnas impares y finalmente filas impares y columnas pares.

Sin embargo, esto da lugar a problemas. Surgen artefactos visuales cuando una partícula se mueve fuera de la región que se estaba procesando. A continuación se muestra un ejemplo de este problema. Cada imagen es una generación de la simulación.

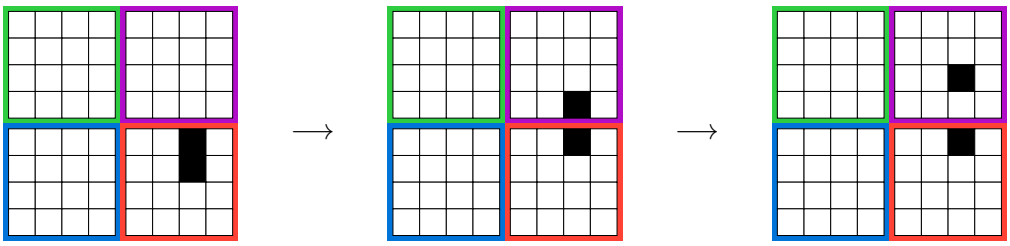


Figura 19: Problema de multithreading

En la Figura 19 se muestran tres generaciones procesando con el sistema multithread descrito. Se ilumina el borde de cara región para mayor claridad. Cada subregión procesa las particulas de arriba a abajo y de izquierda a derecha. El comportamiento de la partícula es el siguiente: Si el vecino superior es vacío, se «mueve» hacia arriba. En tercera generación se puede observar como las partículas se separan. Al haberse procesado primero la región roja, la partícula no puede moverse porque en la región morada había una partícula encima. Acto seguido, la partícula morada se mueve hacia arriba. En ejecución este efecto es notorio y afecta al comportamiento esperado de la simulación. Cambiar el orden en que se actualizan las partículaas resolvería el problema para partículas que se muevan en una dirección determinada, pero el problema siempre se presentará en una dirección.

Para evitar este problema se requirió modificar la actualización de la simulación. En primer lugar, se introdujo y doble buffer, esto permite que el procesamiento de las partículas sea consistente por lo mencionado en la Sección 8.2. Con todo, esto no es suficinete y existen casos específicos en los que el problema persiste. Por ello, además de añadir doble buffer, el orden

en que se actualizan las particulas cambia en un ciclo de 2 fotogramas. Primero se actualiza la matriz de derecha a izquierda y de arriba a abajo, y luego de izquierda a derecha y de abajo a arriba. Esto se debe a que si siempre se actualizan las partículas de izquierda a derecha o viceversa, el sistema presentará un sesgo en la dirección en la que se actualizan las partículas que provoca que el resultado tras varias iteraciones no sea el esperado. Debido a que este efecto se debe a una acumulación de resultados a lo largo de distintas generaciones, se omitirá una explicación detallada. La Figura 20 muestra la diferencia entre actualizar las partículas variando o no el orden de actualización.

(a) Sin variar el orden de actualización (b) Variando el orden de actualización

Figura 20: Diferencia entre variar y no el orden de actualización al procesar partículas

Con estas mejoras el sistema funciona correctamente en casi todos los casos, pero aún existen casos muy específicos resultados de procesar chunks en un patrón de ajedrez. La solución para esto fue variar el orden de actualización de dichos «chunks» o trozos. En cada generación se invierte el orden de actualización de los chunks de una manera simlar a la que se invierte del orden de actualización de las partículas dentro de dichos chunks. Esto permite que el sistema sea más estable y no presente sesgos en la dirección en la que se actualizan las partículas.

Finalmente, para gestionar el procesamiento de los chunks se implementó la técnica del `work stealing`. El hilo principal va asignando chunks a los hilos libros hasta que todos han sido procesados, lo cual deja a los hilos esperando para la siguiente generación.

El procesamiento de una partícula tiene una segunda fase. Una vez todos los chunks han sido procesados, se actualizan los buffers y se actualiza la textura que posteriormente se renderiza en pantalla. La actualización de los buffers consiste en copiar el buffer de la generación actual al buffer de la generación anterior. Esto se hace debido a que hay partículas que podrían no realizar ninguina acción y por tanto no modifican el buffer de la generación actual, por lo que intercambiarlos no es suficiente.

11.4 Simulador en la web

Para poder mencionar Vue, Blockly, WebAssembly y GitHub Pages necesito un capitulo de estado del arte para explicar dichas tecnologías y luego simplemente referenciarlas aquí. Este texto es un recordatorio para mi. Por ahora asume que dicho capítulo existe y cualquier cosa que referncie a aquí es porque ya se ha explicado en dicho capítulo.

La siguiente implementación es distinta a las demás en dos aspectos. Esta se ejecuta en el navegador y además permite a los usuarios definir las reglas de las partículas mediante Blockly. Se profundizará de esto más adelante.

Blockly puede usarse para generar código en cualquier lenguaje, incluido JavaScript, el lenguaje usado en programar elementos interactivos en las webs. No obstante, JavaScript es un lenguaje interpretado que aún siendo JIT, es lento. Debido a esto, desde hace varios años los navegadores tienen soporte para WebAssembly [39], un lenguaje de bajo nivel que es más rápido que JavaScript. Las mayores diferencias entre WebAssembly Y JavaScript, es que WebAssembly es un lenguaje de tipado estático y además, no posee gestión automática de memoria, esta debe ser manejada manualmente.

WebAssembly no está pensdo para ser usado directamente, sino que es un destino de compilación para otros lenguajes. En este caso, se usó Rust, un lenguaje de programación de propósito general con características de lenguajes funcionales y orientados a objetos. Es un lenguaje con características de bajo y alto nivel.

Rust y WebAssembly por si solo no son suficientes. Para poder visualizar el estado de la simulación en la web se usó Macroquad, una librería de Rust que permite renderizar gráficos en la web, además de gestionar la entrada del usuario.

Ejecutar la simulación en la web incurre en un problema no resoluble, no es posible controlar el número de generaciones que se ejecutan por segundo, ya que esto es controlado por la función `requestAnimationFrame` del navegador. Además, el entorno web impide el uso de multihilo, por lo que la simulación se ejecuta en un solo hilo.

Finalmente, para agrupar todos estos elementos, se creó una página web usando Vue y estilizando con TailwidCSS para crear la interfaz de usuario e implementar BLockly. La Figura 21 muestra la interfaz de la simulación en la web.

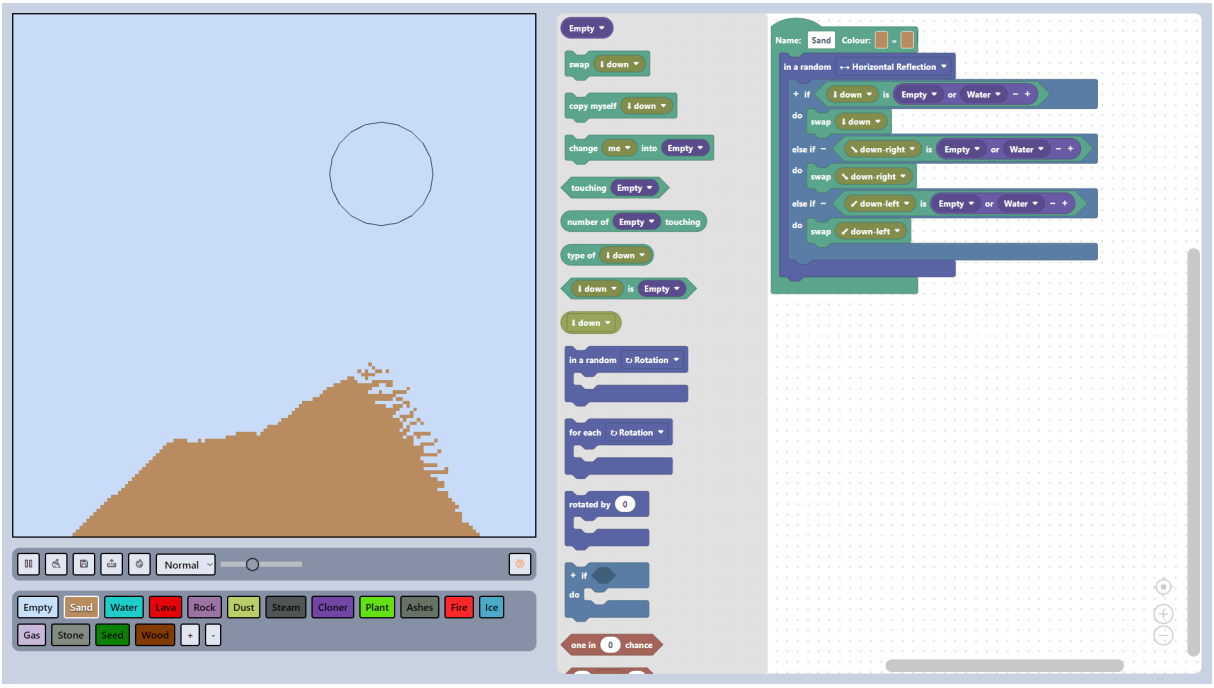


Figura 21: Interfaz de la simulación en la web

A continuación se explica la lógica y funcionamiento interno de la simulación. Esta es una simulación secuencial que no usa doble buffer, pero sí altera el orden de actualización de las partículas para evitar el problema del sesgo visto en la Sección 11.3.1. Las partículas en esta implementación son más complejas y ofrecen más posibilidades. Existen dos tipos de datos: los que posee cada partícula y los que son comunes a todas. Existe un registro asociado a cada tipo de partícula que contiene los siguiente datos: nombre, primer color, segundo color. El nombre sirve para identificar a la partícula en Blockly. Para poder hablar de los dos colores primero es necesario describir la información que tiene cada partícula individualmentea a parte de los parámetros básicos de clock e id, poseen otros 6 datos: opacity, color_fade, hue_shift, extra, extra2, extra3. Todos estos campos son números restringidos a un intervarlo entre 0 y 100.

Cuando una partícula se crea en este sistema, se le asigna un `color fade` aleatorio entre 0 y 100. Este valor controla la interpolación entre el primer y segundo color, `opacity` controla la transparencia de la partícula y siempre se inicializa a 100, `hue shift` altera el tono de la partícula y siempre se inicializa a 0, todos los campos `extra` son valores que se inicializan a 0 y el usuario puede usar para representar cualquier cosa.

Al igual que en la implementación anterior, cada tipo de partícula tiene una función asociada cuyo único parámetro de entrada es un objeto API que contiene las funciones necesarias para interactuar con la simulación. Esta función es generada en tiempo de ejecución. Nuestra implementación de Blockly no genera código, sino que genera datos en formato JSON. Este JSON se envía de JavaScript a WebAssembly (Rust) para ser procesado. Cada bloque de Blockly está asociado a una variante de un enum en Rust, además, estas variantes son tuplas que pueden contener parámetros. El fichero JSON se deserializa en dicha estructura para poder ser procesado mejor. Con esta estructura puede generarse una función. Para ello se define una función que devuelve una función anónima. Se usa pattern matching para que cada variante devuelva una función distinta. Algunas variantes contienen instancias de otras variantes, por lo

que en estas se llama de nuevo a la función que devuelve una función anónima en una suerte de recursión. Finalmente la función obtenida se guarda para ser usada posteriormente.

Este procesamiento no es directo, sino que en función de los datos de las tuplas se toman unas u otras decisiones. Existen dos tipos de datos: constantes y dinámicos. Los datos constantes son aquellos que nunca cambian, mientras que los datos dinámicos dependen del estado de la simulación. Al «convertir» las variantes del enum a funciones, esto se tiene en cuenta. Los datos estáticos son capturados por la función anónima que se devuelve, mientras que los datos dinámicos son recalculados dentro de la función que se devuelve. Un ejemplo sería la dirección. La dirección es un enum que tiene dos variantes: CONSTANT([i32; 2]) y RANDOM. Es evidente que la dirección constante no cambia y puede capturarse en la función anónima, mientras que la dirección aleatoria debe ser recalculada en cada iteración. Esta optimización se aplica en cada variante que tenga una dirección como dato.

No sé si debería poner algún diagrama aquí o explicarlo de otra forma. Viendo el código es mucho más evidente lo que está pasando, lo que hacemos no es una locura pero es algo que considero raro, no me suena haber visto algo así antes. Por si a caso no se entiende, voy a ahorrarme el tener que explicarlo en la tutoria

Imagima que tienes esto

```
enum Action
{
  Move(Direction),
  ChangeColor(Color),
  RandomDirection,
  ExecuteOther(Action),
  ...
}
```

y la siguiente función

Antes que nada, lo que en C++ seria

```
auto lambda = [](API* api) { api->move(Direction::DOWN); };
```

en Rust sería

```
let lambda = |api: &mut API| { api.move(Direction::DOWN); };
```

```
impl Action // Esto significa que las funciones de este bloque son del enum Action,
porque sí, en rust puedes definir funciones para los enums.
{
  fn to_function(&self) -> Box<dyn Fn(&mut API)>
  {
    // En estos ejemplos no realizo ninguna optimización de datos constantes o no
constantes, solo es para que tengas una idea mejor de como va esto
    match self
    {
      Action::Move(direction) => Box::new(|api| api.move(direction)),
      Action::ChangeColor(color) => Box::new(|api| api.change_color(color)),
      Action::RandomDirection => Box::new(|api| api.move(api.random_direction())),
      Action::ExecuteOther(action) => Box::new(|api| action.to_function()(api)),
      ...
    }
```

```
    }
}
```

12 Simulador de arena en GPU

Ejecutar una simulación de partículas en la GPU supone un desafío. Como se explicó en la Sección 8.2, y se mencionó en la Sección 11, los simuladores de arena de por sí no son paralelizables, ya que son dependientes del orden de ejecución y cada celda puede potencialmente modificar al resto de celdas de la matriz. Sin embargo, es posible, reescribiendo las condiciones de evolución de las partículas, transformar un simulador de arena en un autómata celular. Para ello, hay que volver del concepto de partícula al de celda. Esto supone que cada celda tiene que poder conocer su próximo estado mediante reglas locales con sus vecinas, y cada celda solo se modificará a sí misma.

Tomando de nuevo el ejemplo de la Sección 8.2 de simulador básico con un solo tipo de partícula, la de arena, se va a mostrar como se puede convertir este simulador en un autómata celular con un comportamiento similar. Cada celda de este autómata celular solo tiene dos estados, vacío y arena.

La partícula de arena necesita tener el siguiente comportamiento:

- Si la celda de abajo esta vacía, me muevo a ella.
- En caso de que la celda de abajo esté ocupada por una partícula de arena, intento moverme en dirección abajo izquierda y abajo derecha si las celdas están vacías.
- En caso de que no se cumpla ninguna de estas condiciones, la partícula no se mueve.

Ahora, tomando cada celda como entidad aislada, se puede lograr un comportamiento similar de la siguiente forma:

- Si la celda es una partícula vacía, comprueba si encima suyo hay una partícla de arena, en cuyo caso, la celda se convierte en una partícula de arena.
- Si la celda es una partícula de arena, solo existen dos opciones:
  - La celda inmediatamente inferior es vacía, por lo tanto puede caer. La celda actual se convierte en vacía.
  - La celda inferior es arena. En este caso, no es posible saber en el mismo paso de la simulación si la celda inferior también puede caer, por lo que la celda no se mueve y se queda como partícula de arena. Nótese que esto provoca que se produzcan artefactos visuales entre partículas al caer, de manera similar a lo que sucedía en la simulación de Lua con multithreading cuando una partícula se movía fuera de la región en la que se estaba ejecutando. Solo que en este caso, se producen entre todas las partículas en todo momento, ya que cada celda es un hilo de ejecución separado. Se puede observar que caen en líneas horizontales con espacios entre ellas.

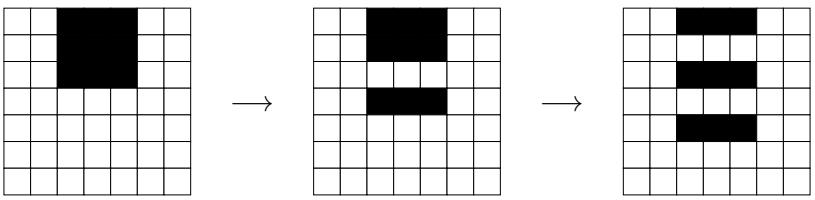


Figura 22: Artefactos visuales

Para realizar movimientos diagonales, se llevan a cabo las mismas comprobaciones que para el movimiento de caída vertical. Sin embargo, en lugar de verificar las celdas ubicadas arriba y abajo, se realizan comprobaciones en las direcciones arriba derecha, abajo izquierda y arriba izquierda, abajo derecha respectivamente.

Desde el punto de vista de desarrollo, para poder ejecutar estas instrucciones de movimiento en la GPU, es necesario programar las reglas de movimiento dentro de un compute shader. Éste recibe como input el estado de la simulación actual y devuelve como output el estado de la simulación tras realizar uno de los tres movimientos.

Cada comprobación de movimiento se realiza de manera separada, es decir, se ejecuta el compute shader de movimiento hacia abajo, y el output de éste lo procesa el compute shader de movimiento diagonal. Es necesario, a su vez, separar la lógica de movimiento abajo derecha y abajo izquierda ya que no sería posible realizar el movimiento hacia ambas diagonales en un solo paso de simulación. Dado el siguiente estado de la matriz:

Figura 23: Ejemplo problema movimiento diagonal

Tanto la partícula de arriba a la izquierda como la de arriba a la derecha pueden moverse a la celda central, sin embargo, la celda central, que es la que tiene que decidir si en el siguiente paso de la simulación existe materia o no en esa posición, no puede saber si alguna de esas partículas se va a mover a esa posición, ninguna de ellas, o ambas.

Esta implementación, a cambio de ser la más rápida en ejecución, como ya se verá en la Sección 13.1, no aporta flexibilidad de ampliación alguna al usuario, ya que el código de lógica de movimiento se ejecuta mediante compute shaders escritos en .GLSL, lo cual es una tarea que puede ser complicada incluso para programadores que no tengan muchos conocimientos de informática gráfica y programacion de GPUs. A su vez, otro problema que presenta esta implementación es que el añadir partículas e interacciones entre ellas requiere mucho más trabajo que sus contrapartes en CPU, ya que requiere transformar las normas de movimiento y de interacciones entre partículas a reglas locales de celdas. La dificultad que presenta ampliar este sistema ha hecho que actualmente solo tenga implementada la partícula de arena.

Se utilizó Vulkano [40], una librería hecha en Rust que actúa como wrapper de Vulkan [41], como librería gráfica para renderizar partículas debido a la flexibilidad y rendimiento que aporta el tener control sobre el pipeline gráfico a la hora del renderizado. Se hizo uso de Bevy [42], motor de videojuegos hecho en Rust, para implementar mecánicas básicas como el bucle principal de juego o procesamiento de input y de Egui [43] para crear la interfaz.

13 Comparación y pruebas

Una vez entendidos los simuladores de arena y las implementaciones realizadas, se procede a evaluar y compararlas de la siguiente forma:

- Por un lado, una evaluación de rendimiento en cada una de las implementaciones, tanto en CPU como en GPU.
- Por el otro, una evaluación de usabilidad. Para esta se evaluará la capacidad de los usuarios de expandir el sistema dadas unas instrucciones.

13.1 Comparación de rendimiento

Para asegurar que la comparación sea justa, todas las pruebas se han realizado en el mismo hardware en 2 equipos distintos. Además, para medir el rendimiento se aumentará el número de partículas simuladas por segundo hasta que ninguno de los sistemas pueda ejecutar la simulación en tiempo real, esto es, 60 veces por segundo.

Las características de los equipos son las siguientes:

Equipo 1:
- CPU: AMD Ryzen 5 5500
- GPU: NVIDIA GeForce RTX 4060
- RAM: 16 GB GDDR4 3200 MHz
- Sistema operativo: Windows 11

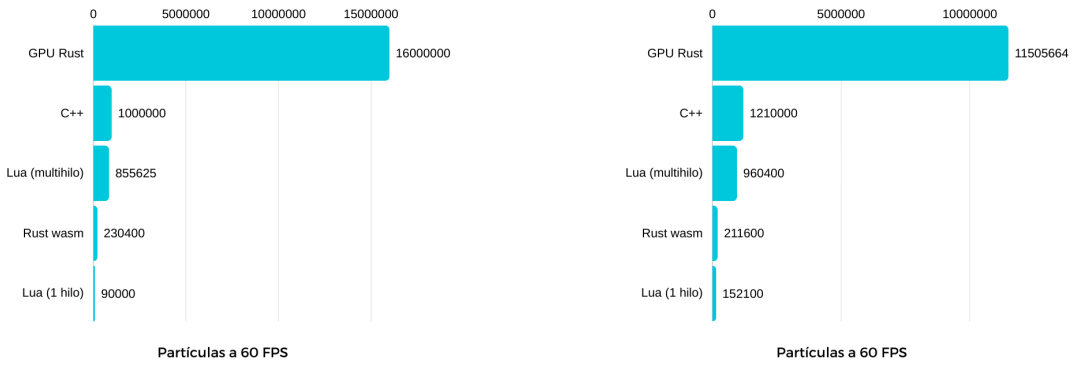
Equipo 2:
- CPU: AMD Ryzen 7 2700x
- GPU: AMD RX 5700XT
- RAM: 32 GB GDDR4 3200 MHz
- Sistema operativo: Windows 11

Se han realizado las diversas pruebas:

- Comparación entre todos los simuladores con el mismo tipo de partícula.
- Comparación entre los simuladores de CPU con una partícula demandante.

Las gráficas estarán ordenadas de mayor a menor cantidad de partículas simuladas a 60 fotogramas por segundo.

A continuación se muestran los resultados obtenidos en las pruebas de rendimiento con GPU:

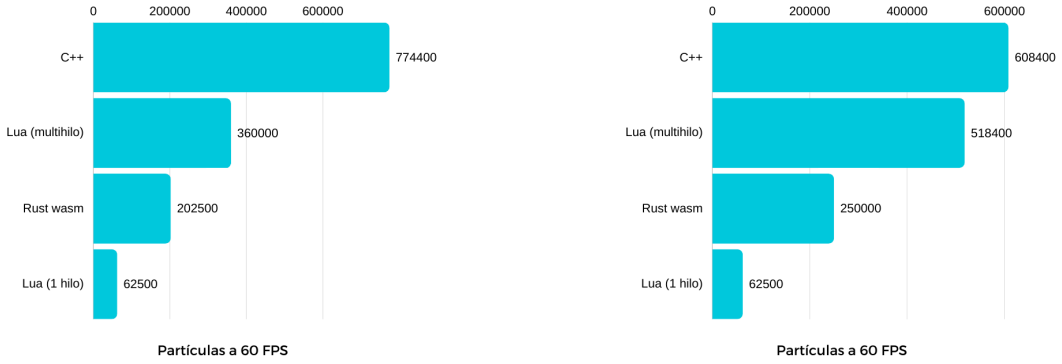


(a) Resultados primer equipo (b) Resultados segundo equipo

Figura 24: Resultados de las pruebas de rendimiento con GPU

Esta solo se usó una partícula de arena, ya que todos los simuladores la implementaban. La implementación de esta es lo más similar posible en todos los simuladores, para que la comparación sea justa. Como puede observarse en la Figura 24, la diferencia entre simular en la GPU y la CPU es considerablemente grande.

A continuación se muestra una segunda prueba, realizada solo entre las implementaciones en CPU. Esto permite observar la diferencia de rendimiento entre los distintos simuladores en CPU mejor que en la gráfica anterior. Además, dado que las implementaciones en CPU tienen más partículas, se ha optado por usar una partícula. Esta partícula tiene la peculiaridad de que necesita comprobar el estado de todos sus vecinos para buscar agua que transformar en planta. Esta búsqueda incurre en un coste computacional mayor que el de la arena, que solo necesita comprobar el estado de sus vecinos para caer.



(a) Resultados primer equipo (b) Resultados segundo equipo

Figura 25: Resultados de las pruebas de rendimiento con GPU

En esta gráfica puede apreciarse la diferencia en cuanto a rendimiento entre ambas implementaciones, además de la diferencia de rendimiento entre una partícula simple y una compleja respecto a la Figura 24

Una vez realizadas las pruebas de rendimiento, se procede a evaluar la usabilidad de los distintos sistemas.

13.2 Comparación de usabilidad

Para evaluar la usabilidad de los distintos sistemas se ha realizado una encuesta a un grupo de 12 personas. En ella se les ha pedido que realicen una serie de tareas en el simulador de Lua, el de Rust web o ambos. Se descartó el simulador en GPU al resultar complejo de expandir y de ejecutar debido a los requisitos necesarios para su ejecución. La tarea fue la misma para ambos simuladores y el proceso fue grabado para su posterior análisis. Se evalúa tanto el tiempo que tardan en realizar la tarea como la cantidad de errores y confusiones que cometen. El grupo de usuarios seleccionado cubre un perfil amplio de individuos, desde de estudiantes de informática hasta personas sin conocimientos previos en programación. En ambos casos, ninguno de los usuarios había utilizado previamente ninguno de los simuladores ni conocían la existencia de los simuladores de arena.

La tarea pedida consistía en crear 4 particulas: Arena, Agua, Gas y Lava. La arena trata de moverse hacia debajo si hay vacío o agua, en caso de no poder, realiza el mismo intento hacia abajo a la derecha y abajo a la izquierda. Es decir, intenta moverse en las 3 direcciones descritas si hay aire o agua. Solo se mueve una vez en la primera dirección en la que es posible en cada generación. La particula de gas tiene el mismo comportamiento que el de arena pero yendo hacia arriba en vez de hacia abajo. La partícula de agua se comporta igual que la arena, pero si no puede moverse en ninguna de las 3 direcciones descritas, se debe intentar mover también directamente a la derecha y a la izquierda, en ese orden. Por último, la particula de lava es igual a la de arena en su movimiento, pero si toca una partícula de agua la convierte en gas. Este punto es importante, pues es la particula de lava la que detecta si hay agua y no al revés. Esto tiene implicaciones en su implementación mediante bloques. Esta tarea es común a las pruebas de Lua y Blockly.

Para la realización de dicha tarea, se explicó que es un simulador de arena y como usarlo. Para esto, se enseña en tiempo real como crear una partícula básica que va hacia abajo sin comprobar nada, además de mencionar como podría hacer la comprobación de detectar una particula en una dirección. Además, se muestra la solución a los usuarios, sin mostrar el código o los bloques, se les enseña las particulas y su comportamiento de forma visual para que tengan una referencia respecto al objetivo a lograr.

Junto con esta memoria se adjuntan los documentos con los guiones realizados para la ejecución de estas pruebas, así como un documento adicional con documentación del simulador de Lua que fue entregado a los usuarios para que pudieran realizar la prueba.

No todas las personas pudieron realizar la prueba de Lua debido a falta de conocimientos o indisposición. Sin embargo otros usuarios que no poseen un perfil técnico accederion e incluso pudieron realizarla. Para minimizar el sesgo, un grupo de usuarios primero realizó la prueba con Blockly y otro con Rust web.

Para la realización de la prueba de Lua, los parámetros registrados son los siguientes:

- Necesitó asistencia en la creación de la lógica: Positivo si el usuario necesitó ayuda activa después de la explicación inicial. Las dudas preguntadas por el usuario no cuentan como necesitar ayuda.
- Usó el método isEmpty: Esta funcion permite comprobar si la particula en una direccion es vacía. Este parámetro es positivo si hizo uso del método sin sugerencia previa.
- Necesitó ayuda para implementar movimiento aleatorio horizontal: Se considera positivo si el usuario necesitó asistencia del probador para añadirle aleatoriedad de movimiento a las partículas
- Necesitó ayuda con errores de ejecución: Se considera positivo si el usuario necesitó asistencia del probador para solucionar la ejecución de una partícula que provocase que el programa crashee.
- Terminó la prueba: Se considera positivo si el usuario terminó la prueba en menos, negativo si por frustración u otras razones no la terminó.

Los resultados son los siguientes:

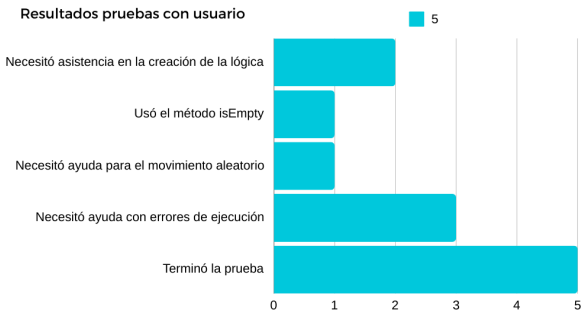


Figura 26: Resultados de las pruebas con usuarios para la prueba de Lua

Todos los usuarios fueron capaces de crear las partículas deseadas de una manera rápida, aunque no de la más efectiva en la mayoría de casos. Prácticamente ningún usuario hizo uso de la función "isEmpty" sin sugerencia previa del probador, la mayoría aprovechaba para reutilizar la lógica creada en la partícula de arena, lo que provocaba que usasen la función "check_neighbour_multiple" de una forma incorrecta, ya que lo empleaban para hacer comprobaciones con un solo tipo de partícula. Todos los usuarios tuvieron errores de ejecución. Las causas más comunes son: falta de escribir "api:" para llamar a funciones de programación de partículas, errores de escritura a la hora de hacer uso de estas mismas funciones, y olvidos de hacer uso de "end" para cerrar claúsulas condicionales "if". La mayoría de participantes pudo implementar aleatoriedad en el movimiento de manera satisfactoria sin ayuda del probador. Como particularidad, un usuario el cual realizó la prueba de Blockly antes que la de Lua, a la hora de implementar aleatoriedad, buscó hacer uso de una funcionalidad "randomTransformation(horizontalReflection)" inexistente en esta simulación. No se detectó ningún sesgo de este tipo por parte de los participantes que realizaron la prueba de Lua y después la de Blockly.

Respecto la prueba de Blockly, los parámetros registrados son los siguientes:

- Necesitó ayuda: Positivo si el usuario necesitó ayuda activa después de la explicación inicial. Las dudas preguntadas por el usuario no cuentan como necesitar ayuda.
- Usó el array: Se refiere a si el usuario uso la funcionalidad de array del bloque de partícula. Este bloque es el primero que se explica y se destaca su función de poder representar varias partículas, además de mostrar un ejemplo de esto. Este parámetro es positivo ei el usuario usó el array en alguna ocasión.
- Usó el foreach: Se refiere a si el usuario uso la funcionalidad de foreach del bloque de partícula. Este bloque se considera el más complejo de entender debido a que las direcciones que están dentro de este son modificadas. Este bloque es necesario para realizar la última tarea. Este parámetro es positivo si el usuario usó el foreach en alguna ocasión sin ayuda.
- Usó una transformación horizontal: Todas las partículas que se pedían tenían la peculiaridad de tener que comprobar ambas direcciones horizontales. Este parámetro es positivo si el usuario usó una transformación horizontal en alguna ocasión sin ayuda. Se espera que ningún usuario use este bloque debido a que no es necesario y no se pide, pero usarlo sería ideal.

- Usó el bloque touching: Este bloque permite resolver el último comportamiento de una forma alternativa. Si el usuario propone o decide usar este bloque, se considera positivo.
- Terminó la prueba: Se considera positivo si el usuario terminó la prueba en menos, negativo si por frustración u otras razones no la terminó.

Los resultados son los siguientes:

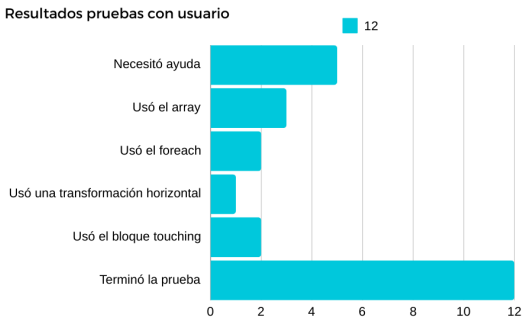


Figura 27: Resultados de las pruebas con usuarios

Además de estos datos cuantitativos, se han recogido datos cualitativos. Estos datos se han recogido durante la prueba mediante anotaciones de los observadores. Los resultados de estas anotaciones son los siguientes:

La mayoría de usuarios presentan problemas al inicio para usar el bloque que representa una partícula, pues a pesar de la explicación y el ejemplo mostrado, la mayoría olvida el funcionamiento de este bloque. Salvo una excepción, todos los usuarios tuvieron problemas para entender el bloque foreach, siendo el bloque que más ayuda necesitó. Por otro lado, se observó que tras una única explicación, los usuarios pueden navegar la interfaz con facilidad y no presentan problemas para añadir, eliminar partículas, poner la simulación en pausa y en general, usar los controles básicos del simulador. Sin embargo, a pesar de que entendían los elementos de la interfaz y su función, hubo uno que causó cierta fricción cognitiva: El botón de añadir partícula. Los usuarios lo pulsaban y procedían a editar bloques en el espacio actual, que no había cambiado, pues los usuarios parecían asumir que añadir una nueva particula la seleccione automáticamente.

14 Conclusiones y trabajo futuro

Una vez concluidas las pruebas, se procedió al análisis de los resultados obtenidos. A diferencia de las pruebas, no se hará una diferenciación entre rendimiento y usabilidad, sino que se decidió evaluar el valor que aporta cada simulador por separado. Finalmente se concluye con una vista global y se proponen mejoras para futuras versiones.

Lo más destacable de las pruebas es la gran diferencia de rendimiento entre los simuladores implementados en CPU y el implementado en GPU. Sin embargo el coste de implementación y extensión es mucho mayor. En este simulador no se pudo probar la usabilidad con usuarios debido a que para ello se requería compilar el proyecto, lo cual requiere ciertas herramientas que muchos usuarios no están dispuestos a instalar. Una implementación en GPU puede resultar ideal cuando se requiere un alto rendimiento pero además el comportamiento que se trata de lograr es altamente específico y de una complejidad moderada. Una implementación de estas características puede resultar útil para la investigación de autómatas celulares en las que se requiera procesar una gran cantidad de partícula simultáneamente para buscar patrones de grandes dimensiones que no podrían ser detectados con una implementación en CPU. Podría resultar interesante la investigación de un sistema que permita generalizar reglas para crear autómatas celulares en la GPU de forma flexible. Esto no ha sido posible con simuladores de arena debido a que en estos, una partícula puede modificar las vecinas, sin embargo, en los autómatas celulares cada celda como mucho puede modificarse a sí misma, lo que podría simplificar la implementación.

La implementación en C++ se realizó como una base para medir las demás. Esta permitió cuantificar la penalización de rendimiento que incurre la flexiblidad usar un lenguaje interpretado como Lua, aún en su versión JIT, así como usar WebAssembly. El desarrollo de simuladores de arena en C++ incurre en el mismo problema que la GPU, se requiere acceso al código y herramientas de desarrollo para poder extenderlo.

En cuanto a la implementacion en Lua, sorprende el rendimiento que puede lograr dada la flexibilidad que ofrece. Sin embargo, desarrollar interfaces es más complejo debido a la escasez de librerías para ello. Con suficiente trabajo, esta implementación tiene el potencial de ser la solución más balanceada de idónea para simuladores de arena en CPU, pues mediante el multihilo el rendimiento logrado resulta ser superior a lo esperado. Si se asume un público objetivo con un perfil más técnico, esta implementación resulta mucho más potente, pues ofrece aún más cotrol que la de Blockly y las pruebas muestran que desarollar partículas programando llega a ser igual de rápido y sencillo que mediante bloques. Para su uso en videojuegos esta opción puede llegar a ser viable con un poco más de trabajo para simular solamente grupos de partículas activas y no la totalidad de las partículas en memoria.

Por último, la implementación en Rust con Blockly destaca por resultar más lenta de lo esperado. La curva de aprendizaje mediante bloques resultó superior a la esperada, sin embargo, pasada esta, los usuarios parecen ser capaces de desarrollar particulas con facilidad sin requerir nociones de programación. Esta implementación resulta ser la más accesible debido a que simplemente requiere de un navegador, software que cualquier dispositivo inteligente actual posee. Debido a su rendimiento, esta implementación no es idónea para explorar simulaciones de una gran complejidad o tamaño. Esto podría paliarse mediante la implementación de multihilo, sin embargo, debido a las reglas de seguridad de memoria de Rust y la poca madurez de multihilo

en WebAssembly, esta tarea resulta en una complejidad muy alta, existiendo la posibilidad de que no se pueda lograr. Por lo tanto esta versión resulta ser la más idónea para simulaciones de baja complejidad y tamaño. También puede considerarse como una opción viable para enseñar lógica e introducir, a los simuladores de arena y, con ciertas modificaciones, a los autómatas celulares.

En conclusión, desarrollar simuladores en GPU resulta ser una buena opción cuando se requiere una gran potencia de cómputo cuando el tamaño de la simulación es muy grande. En CPU, una versión nativa en Lua con multihilo ofrece un rendimiento digno que permite explorar diversos tipos de simulaciones con facilidad siempre que no sean excesivamente complejos. Finalmente, una implementación en Rust con Blockly resulta ser la opción más accesible y sencilla, pero con un rendimiento más limitado.

15 Conclusions and future work

Once the tests were completed, the results were analyzed. Instead of differentiating between performance and usability, the decision was made to evaluate the value provided by each simulator separately. Finally, it concludes with a global view and proposes improvements for future versions.

The most notable thing about the tests is the great difference in performance between the simulators implemented on the CPU and the one implemented on the GPU. However, the cost of implementation and extension is much higher. In this simulator, usability could not be tested with users because this required compiling the project, which requires certain tools that many users are not willing to install. A GPU implementation may be ideal when high performance is required but the behavior to be achieved is highly specific and moderately complex. An implementation of these characteristics may be useful for cellular automata research in which a large amount of particles must be processed simultaneously to search for large-dimensional patterns that could not be detected with a CPU implementation. «It could be interesting to investigate a system that allows for the generalization of rules to create cellular automata on the GPU in a flexible manner This has not been possible with sand simulators because, in these, a particle can modify its neighbors. However, in cellular automata, each cell can at most modify itself, which could simplify the implementation..

The C++ implementation was made as a basis to measure the others. This has allowed us to quantify the performance penalty incurred by the flexibility of using an interpreted language such as Lua, even in its JIT version, as well as using WebAssembly. The development of sand simulators in C++ incurs in the same problem as the GPU, access to the code and development tools are required to be able to extend it.

Regarding the implementation in Lua, the performance it can achieve is surprising given the flexibility it offers. However, developing interfaces is more complex due to the scarcity of libraries for it. With enough work, this implementation has the potential to be the most balanced solution suitable for arena simulators on CPU. Through multithreading the performance achieved turns out to be higher than expected. If a target audience with a more technical profile is assumed, this implementation is much more powerful. It offers even more control than Blockly's, and tests show that developing particles by programming becomes just as fast and simple as using blocks, For use in video games, this option may become viable with a little more work to simulate only groups of active particles and not all of the particles in memory.

Finally, the implementation in Rust with Blockly stands out for being slower than expected. The learning curve through blocks was higher than expected, however, after this, users seem to be able to develop particles with ease without requiring programming notions.This implementation turns out to be the most accessible because it only requires a browser, which is software that any current smart device has. Due to its performance, this implementation is not ideal for exploring simulations of great complexity or size. This could be alleviated by implementing multithreading, however, due to Rust's memory safety rules and the low maturity of multithreading in WebAssembly, this task results in very high complexity, with the possibility that it cannot be achieved. Therefore, this version turns out to be the most suitable for simulations of low complexity and size. It can also be considered as a viable option for teaching logic and introducing arena simulators and, with certain modifications, cellular automata.

In conclusion, developing simulators on GPU turns out to be a good option when great computing power is required when the size of the simulation is very large. On CPU, a native version in Lua with multithreading offers decent performance that allows you to explore various types of simulations with ease as long as they are not excessively complex. Finally, an implementation in Rust with Blockly turns out to be the most accessible and simple option, but with more limited performance.

16 Contribuciones

En este capítulo se detalla la aportación de cada uno de los alumnos en el desarrollo del proyecto.

16.1 Nicolás Rosa Caballero

Existen 3 aportaciones principales de Nicolás al proyecto:

- Simulador en C++
  - Maquetación inicial del proyecto, implementación de OpenGL, GLFW y IMGUI.
  - Implementación de la lógica inicial de la simulación de arena y la interacción con el usuario.
  - Refactorización de las partículas básicas para este modelo: Arena, Agua, Ácido, Roca, Aire y Gas para generalizar su comportamiento y facilitar el desarrollo de nuevas partículas.
  - Investigación e implementación de un sistema de interacciones entre partículas.
  - Optimizaciones y revisiones del código pertinentes para asegurar que la implementación sea un buen referente comparativo para el resto de implementaciones.
- Simulador en Lua con Love2D
  - Diseño de la arquitectura del proyecto y la implementación de toda la lógica de la simulación de arena.
  - Implementación de un sistema de partículas programables en Lua.
  - Diseño e implementación del API para la creación de partículas y sus interacciones.
  - Sistema de eventos implementado con Beholder para disminuir la dependencias entre componentes.
  - Implementación de un sistema de multithreading para mejorar el rendimiento del simulador.
  - Implementación de un algoritmo para encontrar el mayor número de hilos y tamaño de chunk en función del tamaño del canvas y la cantidad de núcleos de la CPU que permita aprovechar la mayor cantidad de hilos simultáneos sin incurrir en condiciones de carrera.
  - Sincronización del trabajo entre los hilos aplicando una implementación propia de la técnica «work stealing» en base al uso de canales.
  - Implementación de un sistema de doble buffer para evitar condiciones de carrera y asegurar la consistencia de los datos.
  - Elaboración de la documentación del API del sistema de partículas.
- Simulador en Rust con Macroquad
  - Creación del proyecto y configuración de las dependencias.
  - Configuración del proyecto para soportar WebAssembly, aplicando características distintas según el destino de compilación mediante flags de compilación condicional.
  - Diseño de la arquitectura del proyecto y la implementación de la lógica de la simulación de arena.

  - Implementación de un sistema de comunicación con WebAssembly mediante una cola de comandos global.
  - Implementación de un sistema de plugins para facilitar la creación de nuevas partículas.
  - Extensión del sistema de plugins mediante la creación de un tipo de plugin que toma como datos un fichero JSON y genera una función a ejecutar para la partícula.
  - Diseño e implementación del formato de JSON para definir partículas y sus interacciones.
- Web Vue3 envoltorio del ejecutable de Rust
  - Creación de la web usando Vue, Vite y TailwindCSS.
  - Diseño en Canva y posterior implementación de la interfaz gráfica de la web.
  - Implementación del código JavaScript pegamento que permite comunicar la web y el ejecutable WebAssembly.
  - Implementación de un sistema de guardado y cargado de plugins en formato JSON.
  - Implementación de un menú de ayuda y un sistema de gestos para facilitar la interacción con la web.
  - Implementación de integración continua en GitHub mediante GitHub Actions para automatizar la generación de la web.
  - Interactividad de los botones, gestos y otros elementos de la web mediante JavaScript y Vue (variables reactivas, watchers).
  - Implementación de Pinia para gestionar un estado global reactivo y minimizar la interdependencia entre componentes.
  - Colaboración con Jonathan para integrar Blockly en la web.
  - Edición de algunos generadores y definiciones de bloques creados por Jonathan para adaptarlos a las necesidades del proyecto.
  - Diseño del logo de la web.
- Otros
  - Elaboración del plan de pruebas con usuario.
  - Realización de parte de las pruebas de usabilidad con usuario.
  - Elaboración de pruebas de rendimiento entre distintos simuladores.
  - Elaboración de figuras para la memoria mediante scripting en Typst y Canva.

16.2 Jonathan Andrade Gordillo

- Simulador en C++
  - Configuración inicial del proyecto así como configurado de solución, proyecto y biblicotecas
  - Implementación de partículas iniciales como agua, roca y gas
  - Asistido en la interaccion con el usuario añadiendo pincel ajustable
  - Añadido propiedades físicas a las particulas como la densidad

- Movimiento que se ajuste a estos parámetros físicos
- Añadido de granularidad a las partículas
- Investigación de sistema alternativo de interaccion entre partículas mediante funciones anónimas
- Solución de bugs a lo largo del desarrollo relacionados con rendimiento e interacciones

- Simulador en Rust con Vulkan haciendo uso de GPU
  - Investigación de posibles formas de hacer uso de la GPU para el cálculo de la lógica, entre ella añadir OpenMP o SYCL al proyecto principal
  - Desarrollo de pipeline gráfico básico haciendo uso de Vulkan
  - Desarrollo de sistema de interacción básico para colocar partículas
  - Implementacion de interfaz mediante ImGUI
  - Implementación de partícula de arena
  - Investigación y desarrollo de compute shaders que permitan delegar el movimiento a la GPU
  - Exploración de diferentes tamaños de work group que den lugar a un mayor rendimiento de ejecución
- Blockly para simulador de Rust
  - Investigación sobre las necesidades del proyecto y los requisitos del módulo de Blockly.
  - Creación de todos los bloques presentes en el proyecto, así como de los posibles mutadores que necesiten a excepcion de uno
  - Ajuste del toolbox para incluir los bloques desarrollados
  - Implementación de los generadores para cada bloque creado, aunque algunos de ellos tuvieron que ser corregidos más tarde junto a Nicolás de
  - Colaboración con mi compañero para incluir Blockly en la página web
- Otros
  - Realización de parte de las pruebas de usabilidad con usuario.
  - Elaboración de pruebas de rendimiento entre distintos simuladores.

Bibliografía

[1] W. Stallings, *Computer Organization and Architecture Designing for Performance*.

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

[3] T. M. Li, *Cellular automata*. Nova Science Pub Incorporated, 2011.

[4] A. Adamatzky, *Game of Life Cellular Automata*. 2010. doi: 10.1007/978-1-84996-217-9.

[5] K.-P. Hadeler y J. Müller, *Cellular Automata: Analysis and Applications*. Springer, 2017.

[6] C. Petzold, *The annotated turing*. John Wiley & Sons, 2008.

[7] J. T. Schwartz, J. Von Neumann, y A. W. Burks, «Theory of Self-Reproducing Automata», *Mathematics of computation*, vol. 21, n.º 100, p. 745-746, oct. 1967, doi: 10.2307/2005041.

[8] J. Von Neumann, «The General and Logical Theory of Automata», *Routledge*, pp. 97-107, jul. 2017, doi: 10.4324/9781315130569-15.

[9] «Conway's Game of Life: Scientific American, October 1970». [En línea]. Disponible en: https://www.ibiblio.org/lifepatterns/october1970.html

[10] S. Wolfram, *A New Kind of Science*. 2002.

[11] Weisstein, Eric W, «Langton's Ant – from Wolfram MathWorld». [En línea]. Disponible en: https://mathworld.wolfram.com/LangtonsAnt.html

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

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

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

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

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

[17] D. A. Patterson y J. L. Hennessy, *Computer Organization and Design*. Morgan Kaufmann, 2013.

[18] T. Möller, E. Haines, y N. Hoffman, *Real-time renderiNg*. A K PETERS, 2018.

[19] J. Krüger y R. Westermann, «Linear algebra operators for GPU implementation of numerical algorithms», 2003.

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

[21] T. M. Aamodt, W. W. L. Fung, y T. G. Rogers, *General-Purpose Graphics Processor Architectures*. Springer Nature, 2022.

[22] S. Chen *et al.*, «Scheduling threads for constructive cache sharing on CMPs», en *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, en SPAA '07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 105-115. doi: 10.1145/1248377.1248396.

[23] J. Gregory, *Game Engine Architecture, Third Edition*. CRC Press, 2018.

[24] Y. Sharvit, *Data-Oriented programming*. Simon, Schuster, 2022.

[25] J. R. Levine, *Linkers and Loaders*. Morgan Kaufmann, 1999.

[26] D. Barron, *The World of Scripting Languages*. John Wiley & Sons, 2000.

[27] R. Ierusalimschy, *Programming in Lua*. Roberto Ierusalimschy, 2006.

[28] C. Ltd, *Mastering lua*. Cybellium Ltd, 2023.

[29] L. H. De Figueiredo, W. Celes, y R. Ierusalimschy, *LUA Programming Gems*. Lua.Org, 2008.

[30] MIT, «App Inventor». [En línea]. Disponible en: https://appinventor.mit.edu/

[31] MIT Corporation, «MIT». [En línea]. Disponible en: https://www.mit.edu/

[32] Google, «Blockly Games». [En línea]. Disponible en: https://code.org/

[33] MIT, «Scratch». [En línea]. Disponible en: https://scratch.mit.edu/

[34] «code.org». [En línea]. Disponible en: https://code.org/

[35] «Blockly Visual». [En línea]. Disponible en: https://developers.google.com/blockly/guides/get-started/workspace-anatomy

[36] «Blockly Block Creation». [En línea]. Disponible en: https://developers.google.com/blockly/guides/create-custom-blocks/define-blocks

[37] D. D. Akinlaja, *LÖVE2d for Lua Game Programming*. Packt Publishing Ltd, 2013.

[38] G. Vault, «https://www.gdcvault.com/play/1025695/Exploring-the-Tech-and-Design». [En línea]. Disponible en: https://www.gdcvault.com/play/1025695/Exploring-the-Tech-and-Design

[39] A. Haas *et al.*, «Bringing the web up to speed with WebAssembly», *SIGPLAN Not.*, vol. 52, n.º 6, pp. 185-200, jun. 2017, doi: 10.1145/3140587.3062363.

[40] «Vulkano». [En línea]. Disponible en: https://vulkano.rs/01-introduction/01-introduction.html

[41] «Vulkan». [En línea]. Disponible en: https://www.vulkan.org/

[42] «Bevy». [En línea]. Disponible en: https://bevyengine.org/

[43] «egui». [En línea]. Disponible en: https://github.com/emilk/egui