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

2 Resumen

Los simuladores de arena, subgénero de autómatas celulares, han experimentado un resurgimiento en popularidad recientemente. Sin embargo, 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

1 Agradecimientos 2
2 Resumen 3
3 Palabras clave 3
4 Abstract 4
5 Key Words 4
6 Introducción 7
6.1 Motivación 7
6.2 Objetivos 7
6.3 Plan de trabajo 8
7 Introduction 9
7.1 Motivation 9
7.2 Objectives 9
7.3 Work plan 9
8 Autómatas Celulares y simuladores de arena 11
8.1 Ejemplos de autómatas celulares 13
8.1.1 Juego de la vida 13
8.1.2 Autómatas de Wolfram 14
8.1.3 Hormiga de Langton 15
8.1.4 Autómata de Contacto 16
8.1.5 Autómata de Greenberg-Hastings 17
8.2 Simuladores de arena 18
8.2.1 Introducción a la simulacion de partículas 18
8.2.2 Importancia y aplicación en diversos campos 18
8.2.3 Simuladores de arena como videojuegos 19
9 Programación paralela 21
9.1 Introduccion historica 21
9.2 Arquitectura GPU 21
9.2.1 Hardware 21
9.2.2 Software 23
9.3 Pipeline gráfico y shaders 24
10 Plug-ins y lenguajes de scripting 26
10.1 Extensión mediante librería dinámica 26
10.2 Extensión mediante un archivo de configuración 28
10.3 Extensión mediante un lenguaje de scripting 29
11 CPU y Multithreading 31
12 Simulador en CPU 33
12.1 Simulador en C++ 33
12.2 Simulador en Lua con LÖVE 33
12.2.1 Multithreading en Lua 34
12.3 Simulador en Rust con Macroquad 36
13 Blockly 38
14 Simulador en GPU 42
15 Comparación y pruebas 44

15.1 Comparación de rendimiento 44
15.2 Comparación de usabilidad 45
16 Conclusiones y trabajo futuro 48
17 Conclusions and future work 50
18 Contribuciones 52
18.1 Nicolás Rosa Caballero 52
18.2 Jonathan Andrade Gordillo 53
Bibliografía 54

Índice de Figuras

Figura 1: Ejemplo de autómata celular sencillo 11
Figura 2: Ejemplo del Juego de la Vida 13
Figura 3: Estructuras estáticas en el juego de la vida 13
Figura 4: Blinker, estructura oscilatoria del juego de la vida 14
Figura 5: Planeador, estructura moviente del juego de la vida 14
Figura 6: Ejemplo autómata de Wolfram 15
Figura 7: Ejemplo simple de la hormiga de Langton 16
Figura 8: Ejemplo completo de la hormiga de Langton 16
Figura 9: Ejemplo de autómata de contacto determinista 17
Figura 10: Imagen gameplay de Noita 19
Figura 11: Comparativa arquitectura de un chip de CPU y de GPU 22
Figura 12: Streaming Multiprocessor 23
Figura 13: Jerarquía de ejecución 24
Figura 14: Generalización de la estructura de un núcleo de CPU 31
Figura 15: Patrón de ajedrez de actualización 34
Figura 16: Problema de multithreading 35
Figura 17: Resultado esperado 35
Figura 18: estructura básica de blockly 39
Figura 19: interfaz blockly pixel creator 41
Figura 20: Resultados de las pruebas de rendimiento con GPU 44
Figura 21: Resultados de las pruebas de rendimiento con GPU 45
Figura 22: Resultados de las pruebas con usuarios 47

6 Introducción

6.1 Motivación

Los simuladores de arena fueron un subgénero destacado durante la decada 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 asi 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 mas 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.

Los autómatas celulares[1] 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 cierto 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[2]. En este contexto, el término «vecinos» hace referencia a las celdas que se encuentran adyacentes a una celda específica.

En el caso específico 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[3]. 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 ejepmlo 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` entonces pasa a estar `Encendida` en la siguiente iteración y viceversa. En resumen, los estados alternan periódicamente. A continuación se muestra un ejemplo de este sistema con 3 iteraciones:

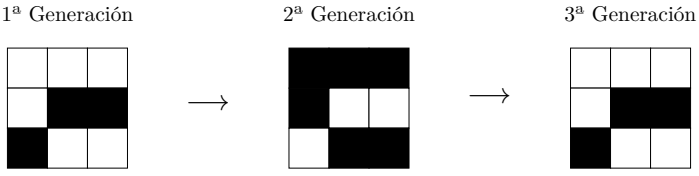


Figura 1: Ejemplo de autómata celular sencillo

Una característica importante de los autómata celulares que los distingue de los simuladores de arena, de los cuales se hablará en la siguiente sección, es su forma de de ser procesados. Esto sistemas no son procesados secuencialmente celda a celda. Se ha de comprender que todas

las celdas «se procesan» a la vez. Esto implica que si una celda cambia de estado, las celdas adyacentes a esta celda no verán su estado modificado hasta la siguiente generación.

Muchos automátas celulares pueden representarse como una máquina de Turing. 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.

Ahora que se ha establecido el marco teórico de los autómatas celulares, es posible explorar su historia y su relevancia en la ciencia y las matemáticas.

El concepto de autómatas celulares tiene sus raíces en las investigaciones pioneras llevadas a cabo por John von Neumann en la década de 1940. Von Neumann, un matemático húngaro-estadounidense, es ampliamente reconocido por sus contribuciones significativas en una variedad de campos, incluyendo la física cuántica, la economía y la teoría de juegos. Su trabajo en la teoría de autómatas y sistemas auto-replicantes[4] fue particularmente revolucionario.

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».

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 investigadoras contribuyeran al desarrollo de los automátas celulares, sin embargo, no fue hasta 1970 que se popularizaron de cara al público general. En este año, Martin Gardner, divulgador de ciencia y matemáticas estadounidense; escribió un artículo[5] en la revista Scientific American sobre un automáta celular específico: El juego de la vida de Conway.

A continuación se mostrarán algunos ejemplos de autómatas celulares relevantes, destacando sus características y aplicaciones en la ciencia y la matemática.

8.1 Ejemplos de autómatas celulares

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

8.1.1 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.

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.

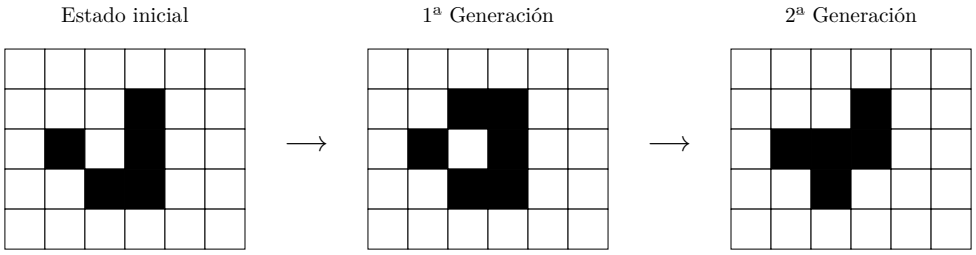


Figura 2: Ejemplo del Juego de la Vida

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 estáticos, como el «bloque», que consiste en un cuadrado de 2x2 células vivas, y el «barco», que se asemeja a una forma de L compuesta por cinco células vivas.

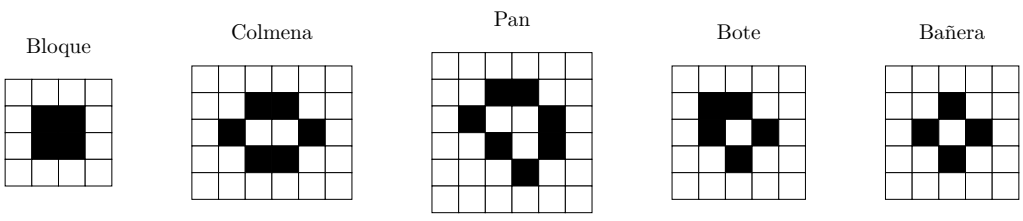


Figura 3: Estructuras estáticas en el juego de la vida

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.

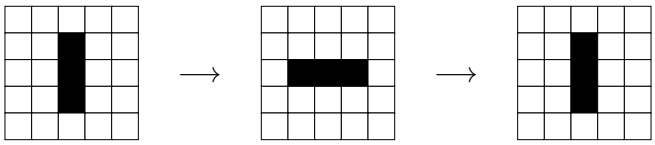


Figura 4: Blinker, estructura oscilatoria del juego de la vida

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.

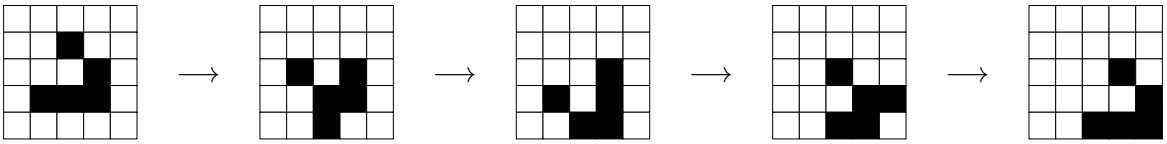


Figura 5: Planeador, estructura moviente del juego de la vida

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 cualquier computadora de Turing). 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 Autómatas de Wolfram

Los Autómatas de Wolfram[3], 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 30^6 de Wolfram tras ejecutar 15 iteraciones de este:

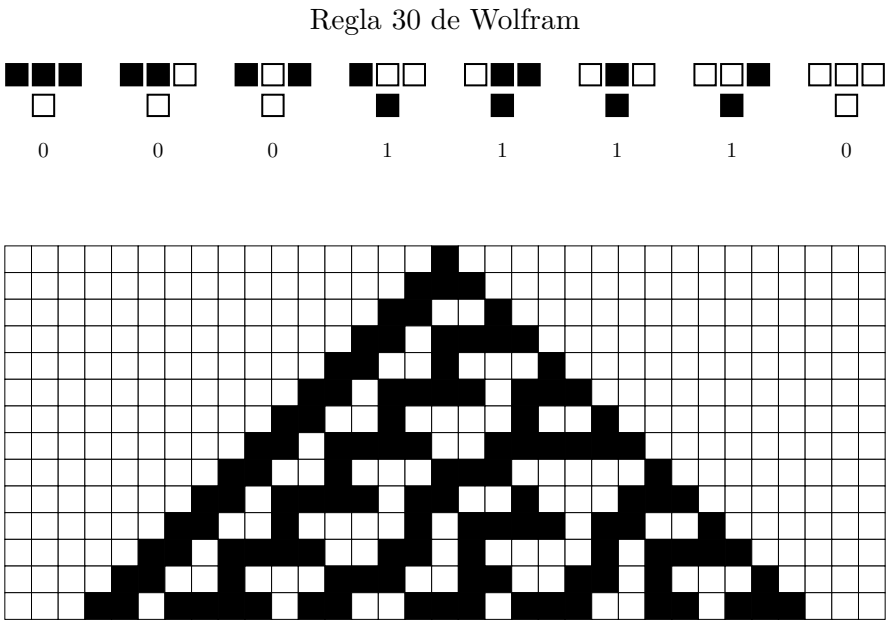


Figura 6: Ejemplo autómata de Wolfram

8.1.3 Hormiga de Langton

La «hormiga de Langton»[3,7], 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[3]. 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.á negra.

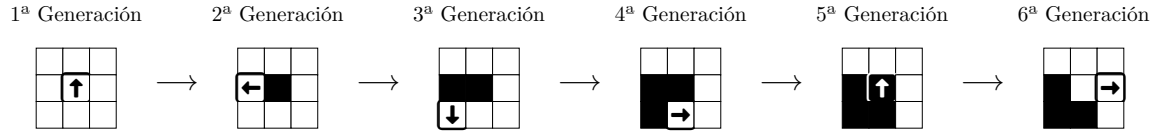


Figura 7: Ejemplo simple de la hormiga de Langton

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:

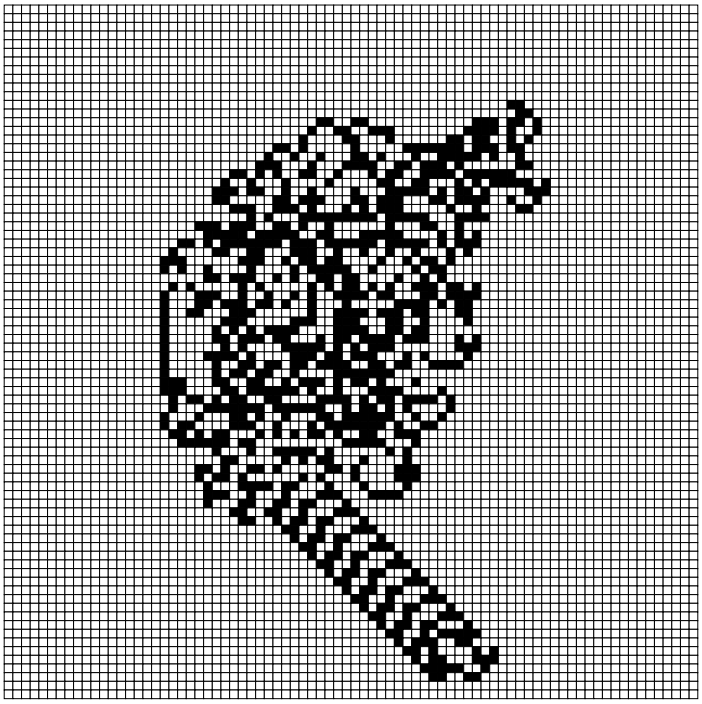


Figura 8: Ejemplo completo de la hormiga de Langton

8.1.4 Autómata de Contacto

Un autómata de contacto[3] puede considerarse como uno de los modelos más simples para 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. La forma general sigue siendo cuadrática. 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.

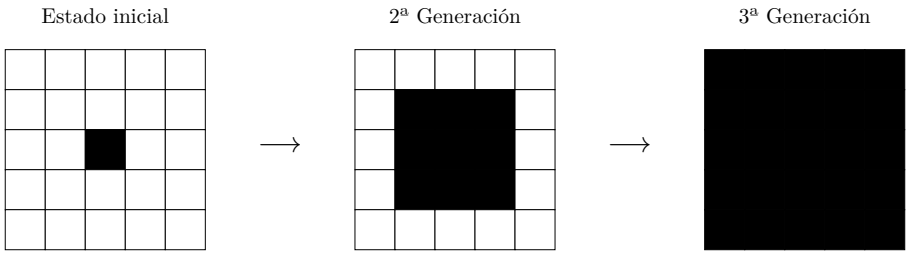


Figura 9: Ejemplo de autómata de contacto determinista

8.1.5 Autómata de Greenberg-Hastings

Los autómatas de Greenberg-Hastings[3] 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. Esto hace que los autómatas de Greenberg-Hastings sean una herramienta valiosa en la investigación biomédica, especialmente en el estudio de la actividad eléctrica del corazón y otros tejidos que exhiben comportamientos similares.

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

8.2.1 Introducción a la simulacion de partículas

Los simuladores de partículas son un subgénero destacado tanto en el sector de los videojuegos como en el de los autómatas celulares[1], haciendose uso de ellos para herramientas y experiencias capaces de representar interacciones dinámicas presentes en el mundo real.

Los simuladores de partículas cumplen las características básicas que permiten considerarlos autómatas celulares, ya que: el «mapa» de la simulación está formado por un conjunto de celdas dentro de un número finito de dimensiones, cada una de estas celdas se encuentra, en cada paso discreto de la simulación, en un estado concreto dentro de un número finito de estados en los que puede encontrarse y el estado de cada celda es determinado mediante interacciones locales, es decir, está determinado por condiciones relacionadas con sus celdas adyacentes.

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

- Simulador de fluidos[8]: Enfoque dirigido a la replicación del comportamiento de sustancias como pueden ser el agua y diferentes clases de fluidos, lo que implica la simulación de fenómenos como flujos y olas y de propiedades físico químicas como la viscosidad y densidad del fluido.
- Simulador de efectos[9]: Categoría que apunta a la simulación detallada de efectos de partículas,como pueden ser las chispas, humo, polvo, o cualquier elemento minúsculo que contribuya a la creación y composición de elementos visuales para la generación de ambientes inmersivos.
- Simuladores de arena[10]: Conjunto de simuladores cuyo objetivo es la replicación exacta de interacciones entre elementos físicos como granos de arena u otros materiales granulares. Esto implica la modelización de interacciones entre particulas individuales, como pueden ser colisiones o desplazamientos, buscando recrear interacciones reales de terrenos.

8.2.2 Importancia y aplicación en diversos campos

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

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

- Física y ciencia de materiales

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

- Ingeniería y construcción

Se emplean simuladores con el objetivo de prever y comprender el funcionamiento de diferentes estructuras y materiales en el ámbito de construcción antes de su edificación, lo que permite predecir elementos básicos como la distribución de fuerzas y tensiones así como el comportamiento ante distintos fenómenos como pueden ser terremotos. El proyecto OpenSees[11] le proporciona esta funcionalidad a ingenieros para comprar la seguridad en el diseño de sus edificaciones.

- Medicina y biología

Los simuladores de partículas en el ámbito de la medicina permite modelar comportamientos biológicos así como, por ejemplo, imitar la interacción y propagación de sustancias en fluidos corporales, ayudando al desarrollo de tratamientos médicos. Por ejemplo, existe el proyecto SimVascular[12] el cual ofrece herramientas para simular y visualizar flujos sanguíneos en modelos de vasos sanguíneos.

8.2.3 Simuladores de arena como videojuegos

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

Este proyecto toma como principal referencia a «Noita», un videojuego indie roguelike que utiliza la simulación de partículas como núcleo principal de su jugabilidad. En «Noita», cada píxel en pantalla representa un material y está simulado 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 destruyendolo o haciendo que interactúen entre sí.

Figura 10: 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[13]

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

- Powder Toy[14]

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[15]

Este proyecto utiliza la misma base de sus sucesores, proporcionando al jugador libertad de hacer interaccionar partículas a su gusto. Además, añade elementos presentes en Powder Toy como el viento, aunque la escala de este proyecto es mas limitada que la de proyectos anteriores. De Sandspiel, nace otro proyecto llamado Sandspiel Club[16], 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[17] de Google. Además, similar a otros títulos menos relevantes como Powder Game (No confundir con Powder Toy), es posible guardar el estado de la simulación y compartirla con otros usuarios. A cambio de esta funcionalidad, en Sandspiel Club no es posible hacer uso del viento, elemento sí presente en Sandspiel.

Hasta ahora se han visto las bases de lo que es un automata celular y los usos y extension de uno de sus subgeneros, los simuladores de arena, además de algunos ejemplos de su aplicación dentro del sector de los videojuegos.

Es intuible el hecho de que la simulación de partículas básicas es un caso de problema “Embarrassingly parallel”, es decir, problema donde la sub-separación de una tarea grande en otras muy pequeñas independientes entre ellas es obvia o directa. Esto, desde el punto de vista de software, permite hacer uso de programación en paralelo para resolver el problema, lo cual abre la posibilidad de utilizar la GPU para procesar la lógica de la simulación. Por lo que ahora, se introducirá al lector al funcionamiento y programacion de GPUs , hablando acerca de su evolución, arquitectura, y sus usos.

9 Programación paralela

9.1 Introduccion historica

Esta sección describe brevemente la evolucion de las GPU's. No se pretende entrar en detalles, sino mas bien señalar los distintos hitos evolutivos que se dieron durante la creación de este hardware.

La necesidad de presentar gráficos en pantalla ha estado siempre presente, desde proyectos como Sketchpad[18] ,animacion 2D cinematográfica o videojuegos, y con ello la necesidad de que los computadores posean hardware que facilite esta tarea.

Durante la evolución historica del proceso de desarrollo de este hardware, podemos diferenciar 3 fases principalmente[19–21]:

En la primera fase, que abarca desde principios de los 80s a mediados de los 90s, la «GPU» constaba de elementos hardware muy especializados y no programables. Cumplian funciones muy específicas de renderizado en pantalla. Por ello, en esta epoca a este tipo de hardware se le denominaba como aceleradores 2D / 3D.

La segunda fase, que duró hasta mediados de los años 2000, abarca el proceso iterativo de la transformacion de una arquitectura fija y muy especifica a una que permitia la programación de shaders dentro del pipeline gráfico, mediante la modificacion de los shaders de vertices y de pixeles. Este hecho es importante, ya que dada la potencia y características de la GPU, que no dejaba de ser hardware destinado al procesamiento gráfico, añadida a la capacidad de personalizar estos shaders, muchos desarrolladores buscaron formas de mapear datos ajenos a los gráficos a texturas y ejecutar operaciones sobre estos datos mediante shaders[22] .El fin de esta era viene marcado por la introducción por parte de NVIDIA de su novedoso lenguaje GPGPU (General-Purpose Graphics Processing Unit), CUDA (compute unified device architecture), capa de software que proporciona acceso directo al conjunto de instrucciones virtuales de la GPU para la ejecución de kernels.

La tercera fase abarca desde el final de la segunda fase hasta la actualidad, y se trata de un proceso de adaptacion de la GPU a tareas no necesariamente relacionadas con el procesamiento gráfico, como puede ser la introducción de compute shaders, shaders que actúan fuera del pipeline gráfico, para evitar que los desarrolladores tengan que aprovechar shaders como el de vertices o de pixeles para realizar operaciones no requeridas para renderizar, o hardware nuevo destinado a tareas de inteligencia artificial como los Tensor Cores, cores incluidos a partir de su serie Volta de GPUs que estan destinados a asistir en el cálculo de operaciones requeridas por procesos de Inteligencia Artificial, como por ejemplo el entrenamiento de redes neuronales .

9.2 Arquitectura GPU

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.

9.2.1 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].

Gran parte de estas son divisibles en tareas más pequeñas. Por ejemplo, la multiplicación de matrices se descompone en operaciones vectoriales, donde el resultado de cada elemento de la matriz se calcula mediante la multiplicación y suma de vectores. Esto permitiría, en caso de tener núcleos con capacidad de cómputo suficientes, el cálculo de cada elemento de la matriz por separado. Este enfoque permite una mayor eficiencia en el procesamiento, lo que resulta en una aceleración significativa en la tarea de renderizado.

Por lo tanto, surge la GPU como co-procesador con una arquitectura SIMD (single instruction multiple data) cuya funcion es la de facilitar a la CPU el procesado de tareas relacionadas con lo gráfico, como renderizar imágenes, videos, animaciones etc[23].

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(recuentode transistores) dentro del chip con respecto a la CPU, que esta diseñada para procesar las instrucciones secuencialmente[19].

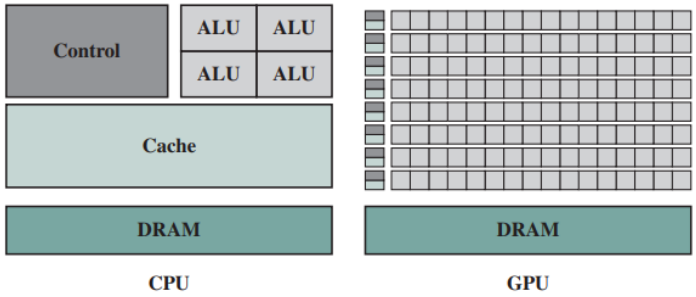


Figura 11: Comparativa arquitectura de un chip de CPU y de GPU

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[19].

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 mas 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 esta ejecutandose[19].

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. Además de núcleos de procesamiento, estas agrupaciones incluyen normalmente una jerarquia 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]

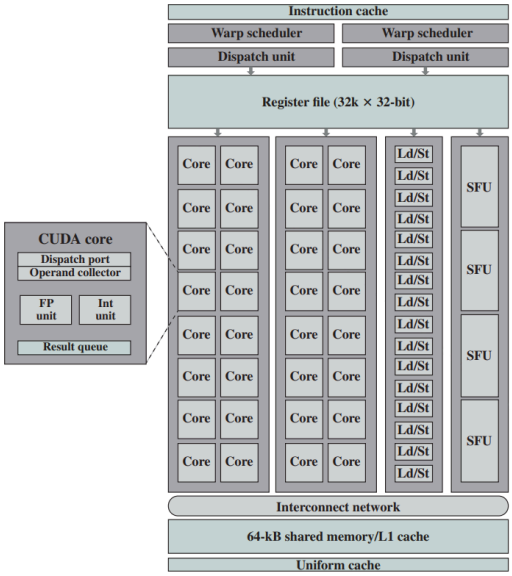


Figura 12: Streaming Multiprocessor

9.2.2 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[19]:

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

Al código destinado a ser procesado por la GPU se le llama kernel. El código que constituye un kernel tendrá la menor cantidad de codigo condicional posibles preferiblemente estar formado por una secuencia de instrucciones definida. A cada instancia de ejecución del kernel se le conoce como hilo. El desarrollador define cual es el numero de hilos sobre los que quiere ejecutar el kernel, idealmente maximizando la paralelización de los calculos. 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.

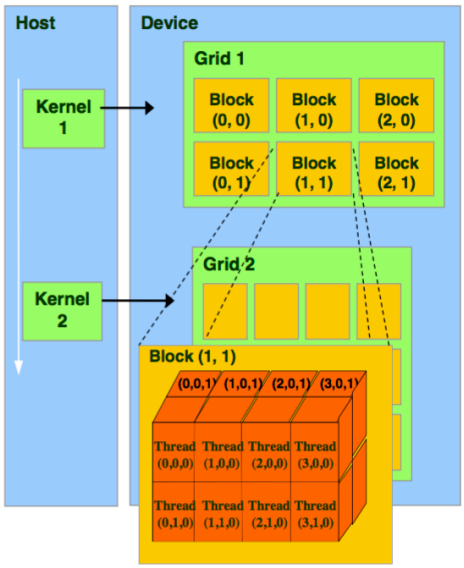


Figura 13: Jerarquía de ejecución

A la hora de ejecutar el kernel, el scheduler global crea warps, sub-bloques de un cierto numero de hilos consecutivos sobre los que se llevara a cabo la ejecucion, que luegos serán programados para ejecutar por el scheduler de cada SM. Es totalmente imprescindible que estos bloques de hilos puedan ser ejcutados de manera totalmente independiente y sin dependencias entre ellos, ya que a partir de aquí el programador de tareas es el que decide que y cuando se ejecuta. Los hilos dentro del bloque pueden cooperar compartiendo datos y sincronizando su ejecución para coordinar los accesos a datos 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.

9.3 Pipeline gráfico y shaders

La función inicial y principal de las GPU era la de procesar la imagen que iba a ser mostrada al usuario. Para facilitar esto, se crearon los shaders, pequeños programas gráficos destinados a ejecutarse en la GPU como parte del pipeline principal de renderizado, cuya base es transformar inputs en outputs que finalmente formarán la imagen a mostrar[24].

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.

Fuera de este pipeline gráfico, existen los compute shaders[25], que son programas diseñados para ejecutarse en la GPU pero que no están directamente relacionados con el proceso de renderizado de imágenes. En lugar de eso, los compute shaders se utilizan para realizar cálculos para otros propósitos que se benefician de la ejecución paralelizable que ofrece la GPU, lo que los hace ideales para tareas como simulaciones físicas, procesamiento de datos masivos o aprendizaje automático.

Su modo de ejecución es el mismo que el mencionado anteriormente usando CUDA, los compute shaders operan en grupos de hilos, lo que permite que múltiples instancias del shader se ejecuten simultáneamente tomando diferentes datos como input.

10 Plug-ins y lenguajes de scripting

Algunos programas necesitan ser extendidos para añadir funcionalidad sin tener que acceder al código fuente. Además, esto permite a usuarios que no sean el desarrollador original añadir funcionalidad al programa. A estos sistemas de extensión se les denomina comúnmente plugins. Un plugin[26] es un componente de software que añade una funcionalidad específica a un programa. Los plugins pueden ser utilizados en programas de todo tipo, desde editores de texto hasta videojuegos.

Existen 3 formas principales de extender un programa: mediante librerías dinámicas, mediante un archivo de configuración o mediante un lenguaje de scripting. Cada una tiene sus ventajas y desventajas. En este capítulo se explorarán y explicarán estas opciones.

10.1 Extensión mediante librería dinámica

Una librería dinámica, también conocida como biblioteca compartida, 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 puede ahorrar espacio en el disco y en la memoria, ya que varias aplicaciones pueden compartir la misma copia de la librería en lugar de tener que incluir su propio código duplicado.

Las librerías dinámicas son especialmente útiles para proporcionar una interfaz estandarizada a funciones de sistema o a librerías de terceros que pueden cambiar con el tiempo. Al vincular estas funciones en tiempo de ejecución, un programa puede beneficiarse de las actualizaciones y correcciones de errores en la librería sin necesidad de ser recompilado. Al mismo tiempo, si el programa está preparado para ello, las librería dinámicas pueden extender funcionalidad.

Las librerías dinámicas, al ser módulos de código que se cargan y vinculan en tiempo de ejecución, necesitan una forma de comunicarse con el programa principal. Esta comunicación implica saber cómo se llaman las funciones, cómo se pasan los datos y cómo se maneja la memoria. En otras palabras, necesitan un protocolo común que permita al programa y a la librería interactuar de manera efectiva. Aquí es donde entra en juego la Interfaz Binaria de Aplicación, o ABI.

La ABI[27] es un contrato entre dos piezas de código binario que define cómo deben interactuar entre sí. Incluye detalles como el formato de los datos, la disposición de las llamadas a funciones y el manejo de la memoria.

La ABI es esencial para las librerías dinámicas porque define cómo el código en la librería debe ser llamado y cómo debe interactuar con el código que la llama. Si la ABI de una librería cambia, cualquier código que dependa de ella puede dejar de funcionar correctamente a menos que sea recompilado para usar la nueva ABI. Por esta razón, los desarrolladores de librerías dinámicas suelen esforzarse por mantener la compatibilidad con la ABI para evitar romper los programas existentes.

Cabe destacar que 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.

Existe un tercer uso de las librerías dinámicas: Organizar código y optimizar tiempos de compilación. Dividir el código en módulos permite compilar solo aquellos que han sido modificados, lo que puede acelerar el proceso de compilación. Sin embargo, esta separación puede ser un arma de doble filo, ya que puede complicar la gestión de dependencias y la resolución de problemas de enlazado. Un ejemplo habitual son los "ensamblados" en .NET, que permiten dividir el código en módulos independientes que se cargan en tiempo de ejecución. Un ensamblado es simplemente una un proyecto de .NET compilado en un archivo DLL o EXE.[28]. Motores como Unity permiten usar distintos ensamblados, además de que es posible definir ensamblados específicos par cada plataforma.

Por último, cabe mencionar que en Windows, las librerías dinámicas pueden contener recursos como texturas, modelos 3D, sonidos, etc. Esto permite, por ejemplo, compartir mods de un juego en un solo archivo.[29].

Sin embargo, las librerías dinámicas tienen algunos problema o inconveniencias a tener en cuenta. Por un lado, la compatibilidad entre plataformas. Cada sistema operativo gestiona las librerías dinámicas de una forma disinta[30], en Windows se usan DLLs, en distribuciones Linux se usan SOs, en MacOS se usan dylibs, por lo que hay que mantener más código. Por otro lado, la compatibilidad entre versiones y el ABI. Además, existen también problemas de seguridad, pues una librería dinámica puede ser modificada por un atacante para lograr la ejecución de código malicioso cierto grado de acceso al sistema. O simplemente un usuario malicioso puede distribuir una librería dinámica que no haga lo que dice hacer.

Mantener la compatibilidad con el ABI también es un problema. Es posible no usar un ABI estable, pero en dicho caso, compilar el programa con una versión distinta del compilador podría dar lugar a binarios no compatibles con librerías dinámicas creadas anteriormente. Esto puede suponer un problema en ciertos sistemas y para la escena de modding, ya que los usuarios podrían dejar de crear contenido si cada actualización del juego rompe los mods. Una opción para evitar esto es usar el ABI de C, que es estable y compatible con la mayoría de lenguajes, pero esto implica no poder usar ciertas características de los lenguajes modernos.[31]. Además puede llegar a dificultar el desarrollo e incluso el rendimiento si no se se implementa correctamente.

La mayor ventaja de este sistema es su flexibilidad. Si el API (las funciones que define la librería) está bien diseñada, las posibilidad de extensión pueden llegar a ser infinitas, véanse los motores de videojuegos, que al final son programas en los que los desarrolladores añaden funcionalidades mediante plugins, normalmente librerías dinámicas. Además, una librería dinámica puede tener estado, es decir, sus propios datos. Este estado es global pero ofrece más posibilidades de extensión y mantenimiento. Por ejemplo, aprovechando que las librerías pueden detectar cuando se carga y se descarga, sería posible para un plugin guardar el número de veces que se ha cargado, para ello lo único que hay que hacer es escribir en un fichero la cantidad de veces que se ha cargado y leerlo al cargar la librería. Extendiendo este sistema, un plugin podría persistir

todo su estado en un fichero. Esto permite a los sistemas de mods gestionar su propio estado sin tener que depender del programa principal.

10.2 Extensión mediante un archivo de configuración

Un archivo de configuración es una herramienta que ofrece la posibilidad de ajustar el comportamiento de un programa sin necesidad de modificar su código fuente y `sin necesidad de compilar código`. Esta práctica resulta especialmente útil en situaciones donde acceder al código no es una opción viable, ya sea porque no está disponible o porque el usuario carece del conocimiento técnico necesario para modificarlo, o porque el desarrollador o empresa quiera mantener el control sobre el código fuente.

La lectura de este archivo por parte del programa le permite realizar acciones específicas basadas en la información contenida en el mismo. Es como proporcionarle al programa un conjunto de instrucciones adicionales que le dicen cómo comportarse en ciertas circunstancias.

Una de las ventajas más destacadas de este enfoque es que es multiplataforma. Debido a que es un formato específico creado a medida para el programa, no depende de ningún protocolo que indique como cargar símbolos (nombres de funciones) o como gestionar la memoria y datos. Esto facilita la mantención del código.

Además, el uso de archivos de configuración no presenta riesgos de seguridad inherentes. A diferencia de ejecutar código arbitrario, que puede ser potencialmente peligroso, la lectura de un archivo de configuración solo implica que el programa interprete ciertos datos de manera específica. Al mismo tiempo, este es el mayor problema de este foramto. En una librería dinámica, el desarrollador puede realizar las operaciones que quiera dentro de una función, siendo la única limitación las funciones que el programa principal exponga para comunicarse con este. En el caso de un fichero de configuración las limitaciones se extreman al no poder controlar nada que no esté especificado en el formato. Es como si tuvieras una serie de piezas que puedes combinar de forma específica, porque el diseñador de las piezas las creó de tal forma que no todas encajan con todas, además se está limitado a las piezas que el diseñador creó. Por otro lado, al definir una serie de órdenes secuenciales que no pueden conocerse de antemano, el código no puede optimizarse tanto como en una DLL, pues hay que leer el archivo e interpretarlo adecuadamente. Sin embargo las DLLs, una vez cargadas podían ejecutar código directamente.

Los archivos de configuración, al ser de especificación libre y no estar sujeto a normas, pueden no ser ficheros de texto. Es posible que sean un formato binario no legible para el ser humano. Sin embargo, lo más habitual es que sean ficheros de texto, ya que la intención es que sean fácilmante modificables por el usuario. Además, que sea un fichero de texto elimina la dependencia de una herramienta que pueda leer y escribir el formato del archivo.

Como ejemplo se muestra un archivo de configuración para un sistema bullet hell inspirado en Danmaku:

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

```
8                                   <0,0.4>
9                                   <0,0.7>
```

Un sistema bullet hell es un tipo de juego de disparos en el que el jugador debe esquivar una gran cantidad de proyectiles en pantalla. En este caso, el archivo de configuración define un patrón de disparo que lanza proyectiles azules en forma de círculo hacia arriba. El patrón se repite cada 8 fotogramas y los proyectiles se mueven aleatoriamente en el eje X entre -6 y 6 unidades. Cada proyectil tiene una velocidad inicial de 0.4 unidades en el eje Y y aumenta a 0.7 unidades a lo largo de su vida útil.

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

10.3 Extensión mediante un lenguaje de scripting

Existe un punto intermedio en cuanto a ventajas y desventajas entre los sistemas ya mencionados: los lenguajes de scripting. Un lenguaje de scripting 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, lo que significa que el código se ejecuta directamente por un intérprete en lugar de ser traducido a código máquina antes de la ejecución.

Un lenguaje de scripting, al igual que otros lenguajes de programación, posee la característica de ser Turing completo. Esto implica que tiene la capacidad de llevar a cabo cualquier operación computacional que sea teóricamente posible. Sin embargo, debido a que los lenguajes de scripting son interpretados en tiempo de ejecución, en lugar de ser compilados previamente, pueden presentar una velocidad de ejecución más lenta en comparación con los lenguajes compilados.

La interpretación en tiempo de ejecución significa que el código se traduce a código máquina mientras el programa se está ejecutando, lo cual puede introducir una sobrecarga adicional que afecta el rendimiento. A pesar de esta desventaja en términos de velocidad, los lenguajes de scripting ofrecen la ventaja de ser multiplataforma.

Ser multiplataforma significa que los scripts escritos en un lenguaje de scripting pueden ser ejecutados en diferentes sistemas operativos o plataformas de hardware. Esto es posible porque el intérprete del lenguaje de scripting, que es el programa que realiza la traducción a código máquina, suele estar disponible para una amplia variedad de plataformas.

Además, 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 Lula. 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. 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 motore 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.

Existe una versión más rápida de Lua llamada LuaJIT. El nombre no es arbitrario, JIT hace referencia a un término en inglés que significa «Just In Time», es decir, «Justo a Tiempo». Existen dos estrategias de compilación: la compilación AOT (Ahead Of Time) y la compilación JIT (Just In Time). La compilación AOT consiste en compilar el código fuente a código máquina antes de la ejecución del programa. Tradicionalmente se llaman lenguajes compilados a los lenguajes que compilan con esta estrategia. La compilación JIT, por otro lado, consiste en compilar el código fuente a código máquina durante la ejecución del programa. La compilación JIT tiene la ventaja de ser multiplataforma. LuaJIT es un intérprete de Lua que utiliza la compilación JIT para mejorar el rendimiento de los scripts Lua. Esto significa que el código Lua se compila a código máquina en tiempo de ejecución, lo que puede mejorar significativamente la velocidad de ejecución de los scripts.

Sin embargo, cabe destacar que en los lenguajes JIT, puede llegar a ser importante saber ciertos detalles de su funcionamietno 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 lenguajes. Por ejemplo, si una función recibe dos parámetros y esos dos parámetros siempre son números, LuaJIT puede optimizar la función en base a heurísticas. 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. De ser del interés del lector, se recomienda leer al respecto el monomorfismo y polimorfismo de JavaScript, ya que suele ser implementarse con una estrategia JIT y existe documentación adecuada al respecto.

A continuación se explicará de forma general qué es una CPU, que es el multithreading o multihilo y cuales son sus ventajas e inconvenientes.

11 CPU y Multithreading

Para poder ejecutar los programas se requiere de una estructura que pueda procesar un conjunto de instrucciones dados, esta estructura es la Unidad Central de Procesamiento (CPU, por sus siglas en inglés). La CPU es el componente principal de un ordenador y otros dispositivos programables, que interpreta y ejecuta instrucciones contenidas en los programas informáticos. La CPU lleva a cabo la mayoría de los cálculos que permiten a un sistema operativo y sus aplicaciones funcionar. La CPU no es una pieza única, sino que está compuesta por una series núcleos que a su vez están compuestos por otras partes[23]. Cada CPU es distinta pero todas presentan al menos los siguientes componentes en la estructura de su núcleo: la Unidad de Control (UC), la Unidad Aritmético Lógica (ALU), varios registros, la Unidad de Punto Flotante (FPU), una memoria de muy rápido acceso llamada caché y un conjunto de buses que conectan estos componentes.

La UC supervisa y coordina las operaciones de la CPU. Controla la secuencia de ejecución de las instrucciones y dirige el flujo de datos entre la CPU y otras partes del sistema. La ALU realiza operaciones aritméticas y lógicas en los datos, como sumas, restas, multiplicaciones, divisiones y operaciones booleanas con la restricción de estar limitada a números enteros. Los registros son pequeñas áreas de almacenamiento de alta velocidad que se utilizan para almacenar datos temporales y direcciones de memoria, estos forman parte del ensamblado principal de la CPU, suelen estar muy cerca de las ALUs. La FPU es una unidad especializada que realiza operaciones en números de punto flotante, como sumas, restas, multiplicaciones y divisiones, suele ser una subparte de la ALU. La caché es una memoria de muy rápido acceso que almacena copias de datos, tienen más almacenamiento que los registros pero son más lentas en comparación y suelen estar un poco más alejadas de las ALUs respecto a los registros. Los buses son caminos de comunicación que permiten que los datos se muevan entre los componentes de la CPU y otros dispositivos del sistema. A continuación se muestra un diagrama generalizado de la estructura de un núcleo de CPU. Cabe destacar que existen distintos tipos de arquitecturas y la siguiente ilustración no pretende represeentar un tipo de CPU en específico sino el concepto general de CPU.

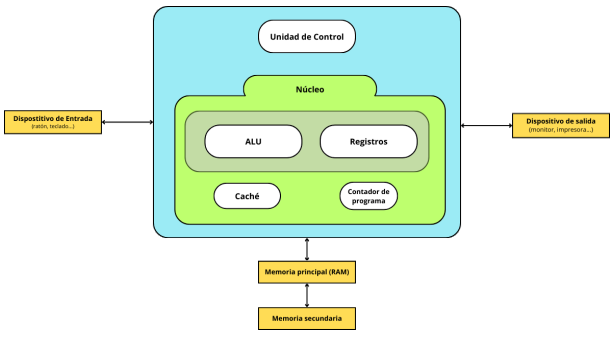


Figura 14: Generalización de la estructura de un núcleo de CPU

Un núcleo de CPU es una unidad de procesamiento que puede ejecutar instrucciones de programa usando los componentes mencionados. En los sistemas modernos, una CPU puede tener múltiples núcleos, lo que permite que se ejecuten múltiples instrucciones simultáneamente.

Los hilos, o threads[23], son una forma de dividir un programa en dos o más tareas paralelas. Cada hilo en un programa ejecuta su propio flujo de control. En un sistema con un solo núcleo„ donde el sistema operativo alterna rápidamente entre ejecutar hilos de diferentes programas.

Es posible aprovechar aún más el rendimiento mediante una tecnología llamada Hyperthreading. Esta, es una tecnología desarrollada por Intel que permite a un solo núcleo de CPU simular dos núcleos lógicos. Como se ha visto, un núcleo tiene distintos componentes, simular dos núcleos lógicos permite usar más de un componente al mismo tiempo. Por ejemplo, si una parte del programa está haciendo una operación con números enteros (usando la ALU), es posible que otra parte de este use la FPU simultáneamente para hacer operaciones con números de punto flotante. De esta forma se aprovechan mejor los recursos de la CPU y se obtiene más rendimiento.

Cada hilo posee su propia región de memoria en el stack, o pila. La pila es una estructura de datos que sigue el principio de «último en entrar, primero en salir» (LIFO, por sus siglas en inglés). Se utiliza para almacenar variables locales y para rastrear el progreso en la ejecución del hilo, como las llamadas a funciones. No obstante, todos los hilos en un proceso comparten el mismo espacio de memoria del heap. El heap es una región de memoria utilizada para el almacenamiento dinámico de datos, es decir, para variables globales y datos asignados dinámicamente durante la ejecución del programa.

La programación paralela es una técnica que se utiliza para mejorar el rendimiento de los programas al permitir que se ejecuten múltiples tareas simultáneamente. Esta técnica se refiere a la capacidad de un programa para ejecutar tareas de manera simultánea en diferentes núcleos o procesadores.

No obstante, la programación paralela presenta desafíos únicos. Entre estos se incluyen problemas de sincronización, donde diferentes hilos intentan acceder o modificar los mismos datos al mismo tiempo. También se pueden presentar problemas de bloqueo o “deadlocks”, donde dos o más hilos se bloquean mutuamente al esperar que el otro libere un recurso.

Sin embargo, dichos problemas tienen soluciones y permiten lograr un incremento de velocidad considerable. Esto será relevante para el siguiente apartado, donde se explicarán las distintas implementeciones de simuladore de arena realizadas en CPU. Posteriormente, se compararán con las implementaciones realizadas en GPU en términos de rendimiento y usabilidad.

12 Simulador en CPU

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 diferencia de los autómatas celulares, estas implementaciones se procesan de forma secuencial. Si una partícula cambia su estado, las demás partículas verán esos cambios reflejos al momento sin tener que esperar a que todas las demás partículas hayan sido procesadas.

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

12.1 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.

Es importante no procesar una misma partícula dos veces. Tanto en esta como en las demás implementaciones, la grilla de partículas se representa como un array bidimensional. Además, en este simulador se actualizar de «arriba a abajo y de izquierda a derecha». Es decir, se procesa primero la primera fila, luego la segunda, y así sucesivamente. Dentro de cada fila, se procesa de izquierda a derecha. Esto provoca que si una partícula se «mueve» hacia abajo, se procesará de nuevo en la siguiente iteración. Para evitar este problema, cada particula tiene un valor llamado clock que tiene dos posibles estados y alterna en cada iteración. Cuando una partícula «se mueve», cambia su valor de clock y no vuelve a ser procesada. Puede pensarse como un marcador que al cambiar indica que dicha partícula fue procesada para la generación actual y no debe volver a procesarse. Es necesario definir que significa que una partícula «se mueva». Esto es una ilusión visual, el sistema ejecuta una transformación de la matriz de partículas en base a unas reglas en pasos discretos de tiempo. Se define «moverse» el proceso por el cual una particula se borra de una celda para escribirse en otra. Esto provoca la sensación de movimiento.

Esta implementación ejecuta una lógica directa en un solo hilo. Debidoo a esto es la base para comparar el rendimiento de las siguientes implementaciones. La siguiente versión a implementar fue desarrollada usando LuaJIT en el framework LÖVE.

12.2 Simulador en Lua con LÖVE

LÖVE es un framework de desarrollo de videojuegos en Lua orientado a juegos 2D. Permite dibujar gráficos en pantalla y gestionar input 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 Lua. FFI significa Foreign Function Interface, es una librería que permite a Lua interactuar con

código 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.

```lua
-- Particle.lua

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

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

Este sistema también usa la técnica de clock para gestionar la ejecución de las particulas. Sin embargo, a diferencia del anterior, el orden en que se actualizan las particulas cambia en un ciclo de 4 fotogramas. Esto permite conseguir un resultado visual más uniforme.

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 su función a ejecutar. Una vez hecho esto, el usaurio solo debe arrastrar su archivo a la ventana de juego para cargar su plugin.

Sin embargo, simular partículas es un proceso intensivo. Además, las particulas no tienen acceso a toda la simulación, debido a esto, se optimizó mediante la implementación de multihilo.

12.2.1 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 exacamente 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. Además de esto, love proveee canales de comunicación entre hilos, que permiten enviar mensajes de un hilo a otro y sincronizarlos. Esto permitió enviar trabajo a los hilos bajo demanda.

La implementación del multihilo dio lugar a problemas que no se tenían antes: escritura simultánea y condiciones de carreras. Esto supuso un desfío 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 grilla en un patron de ajedrez. Esto permite que dos partículas no accedan a la misma casilla al mismo tiempo, ya que cada lote de procesado está separado de los demás para que si la particula se sale, pueda coincidir con otra.

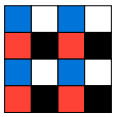


Figura 15: Patrón de ajedrez de actualización

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

pueda interactuar con partículas que no estén inmediatamente cerca sin que acceda a al chunk que está siendo procesado por otro hilo.

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

Además de esto existe otro problema. Podían surgir artefactos visuales cuando una partícula se movía fuera de la región que se estaba ejecutando. A continuación se muestra un ejemplo de este problema. Cada imagen es una generación de la simulación.

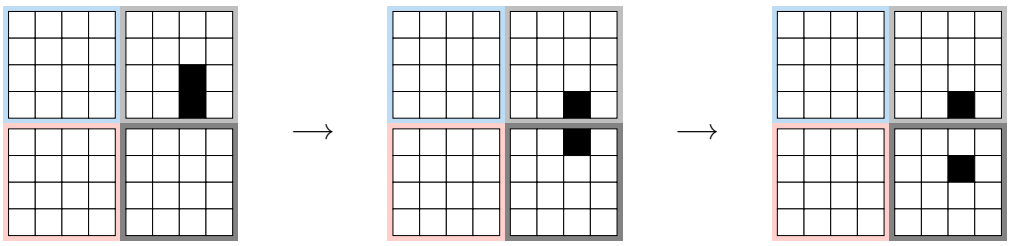


Figura 16: Problema de multithreading

La tercera generacion ya deja ver el problema. Si la simulación fuera single thread, el estado de la simulación sería el siguiente:

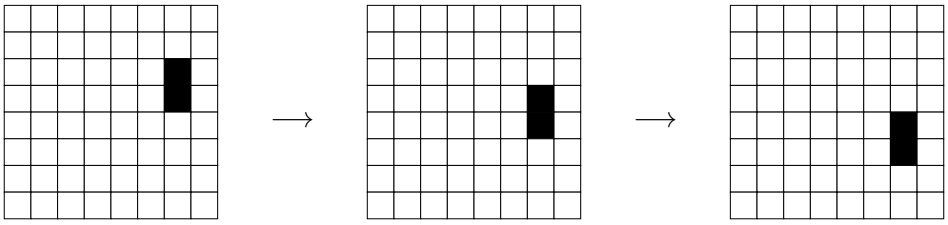


Figura 17: Resultado esperado

Alcanzar la solución esperada en la figura 17 es teóricamente imposible. Sin embargo es posible convertir el comportamiento indeseado en la norma. Para ello se decidió cambiar el procesamiento del sistema a un sistema de doble buffer. Para una mayor comprensión, este sistema puede entender como el usado en los autómata celulares formales. Todas las celdas se ejecutan «a la vez», es decir, si unca celda camnbia así lo dictan las reglas del sistema, las vecinas no percibirán ese cambio hasta la siguiente iteración. Esto permite que el procesimiento sea más uniforme aún cuando la particula se mueve fuera de su área de procesamiento.

Con esto el sistema funciona satisfactoriamente, es suficientemente rápido y fácil de extender mediante mods que son arrastrar y soltar. Existe un API (PDF, documento adjunto, no sé como indicarlo) que explica a los usuarios como crear sus propias particulas. El sistema permite referenciar otras particulas. Es decir, una partícula de arena puede comprobar si debajo de ella hay agua. Esto otorga una gran flexibilidad y portabilidad, además de un rendimiento competente gracias a la compilación `Just in Time` y el aprovechamiento de los recursos de la CPU.

Finalmente, se explora la posibilidad de tener un sistema más eficiente manteniendo la flexibilidad. Para ello, se implementó una versión en Rust con Macroquad.

12.3 Simulador en Rust con Macroquad

Rust es 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, es decir, permite manipular la memoria directamtene pero al mismo tiempo permite programar con atajos y abstracciones que ocultan lógica subyacente para simplificar la tarea del progrmador.

En esta versión se decidió usar Macroquad como librería para poder gestionar la ventana, input y gráficos. Macroquad es un framework de Rust inspirado en raylib. Es multiplataforma y compila de forma nativa a WebAssembly, por lo que el mismo código puede ser usado para la versión web y la nativa. Como desventaja, el game loop viene hecho y no permite cambiar la tasa de refresco, la justificación del desarrollador es que en WebAssembly la tasa de refresco depende de la función de Javascript `requestAnimationFrame` usada para implementar el game loop. A pesar de este inconveniente, Macroquad es muy sencillo de usar y tiene una documentación muy completa.

Para este sistema se implementó una interfaz abstracta de carga de plugins. Esto permite que un plugin pueda ser una librería dinámica, un fichero de configuración o incluso usar un lenguaje de scripting. En este sistema, las partículas son estructuras que tienen un identificador, la variable clock para gestionar su actualización y dos campos extras: light y extra. Light controla la opacidad de la particula en un rango de 0 a 100 mientras que extra es un campo de 8 bits que puede ser usado para guardar información adicional.

Este proyecto se dividió en 3 sub proyectos: plugins, app-core y app. Plugins contiene plugins por defecto para inicializar la aplicación con algo más que una particula vacía, app-core contiene toda la lógica de la simulación y la interfaz abstracta de plugin (traits, en nomenclatura de Rust), app es un crate binario que usa app-core para crear la simulación y renderizarla, es decir, es la aplicación principal.

El objetivo de esta implementación era su uso en una web con integración de Blockly para permitir a los usuarios crear particulas de forma visual. Debido a esto, esta implementación es monohilo, pues no es posible procesar WebAssembly en multihilo.

WebAssembly (también conocido como wasm) es un formato de código binario portátil y de bajo nivel diseñado para ejecutarse de manera eficiente en navegadores web. Es un estándar abierto respaldado por la mayoría de los principales navegadores web, lo que permite ejecutar código escrito en lenguajes de programación de alto nivel, como C++, Rust y JavaScript, en el navegador a una velocidad cercana a la ejecución nativa.

La principal ventaja de WebAssembly es su capacidad para mejorar el rendimiento de las aplicaciones web al permitir la ejecución de código de manera más eficiente en el navegador. A diferencia de JavaScript, que es un lenguaje interpretado, WebAssembly se compila en un formato binario que se puede ejecutar directamente por el navegador, lo que resulta en una ejecución más rápida y eficiente.

Además, WebAssembly es un formato independiente de la plataforma, lo que significa que se puede ejecutar en diferentes sistemas operativos y arquitecturas de CPU sin necesidad de

realizar modificaciones en el código fuente. Esto lo hace ideal para aplicaciones que requieren un alto rendimiento, como juegos, simulaciones y aplicaciones de edición de imágenes.

Sin embargo, WebAssembly también tiene algunas limitaciones. Los tipoos disponibles son limitados y esto complica la comunicación entre WebAssembly y JavaScript.

Como se ha mencionado, es priotiario la flexibilidad y la extensibilidad del sistema. Para ello, se implementó un sistema de plugins que permite a los usuarios crear sus propias particulas y añadirlas al sistema. Para permitir esto, se definió una serie de instrucciones en Rust que pueden ser cargadas de un fichero JSON. Dicho fichero no sería creado manualmente, sino que se generaría a partir de un editor visual de bloques. Este editor visual de bloques se implementó en Blockly, una librería de Google que permite crear interfaces gráficas de programación.

Para poder comunicar WebAssembly con Blockly, se creó una página web usadno el framework Vue3. Vue3 es un framework de JavaScript que permite crear interfaces de usuario de forma sencilla y eficiente. Además, Vue3 es muy flexible y permite integrar otras librerías de JavaScript de forma sencilla. Esto permitió definir una interfaz de usuario amigable y permitir el uso del sistema de bloques definido en Blockly comunicando el binario WebAssembly con el código JavaScript de la página. Debido a que el sistema está preparado para recibir plugins, se implementó en la web un sistema de carga y descarga de lotes de plugins para facilitar el testeo del sistema y al mismo tiempo permitir a los usuarios guardar sus creaciones.

Finalmente cabe destacar que para simplificar el proceso de desarrollo, se optó por alojar la página en GitHub Pages. GitHub Pages es un servicio de alojamiento web gratuito que permite a los usuarios publicar sitios web estáticos directamente desde un repositorio de GitHub. Se configuró un flujo de trabajo de GitHub Actions para automatizar la compilación y despliegue de la página web en GitHub Pages. Esto permite que cualquier cambio en el repositorio se refleje automáticamente en la página web, lo que facilita la colaboración y el desarrollo del proyecto.

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

13 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 codigo 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' del MIT,para crear aplicaciones para Android, 'Blockly Games', que es un conjunto de juegos educativos para enseñar conceptos de programación que también pertenece a Google, 'Scratch', web que permite a jóvenes crear historias digitales, juegos o animaciones o Code.org para enseñar conceptos de programación básicos.

El motivo de la utilización de blockly como biblioteca para desarrollar el comportamiento deseado fue que es la biblioteca mas extendida para realizar este tipo de comportamiento. No existe mucho software que te permita generar codigo personalizado utilizando tus propios bloques. Además, tiene otras ventajas como que es de código abierto, lo que facilitó el desarrollo al permitir consultar el código fuente para revisar fallos o comprender cómo se crean ciertos elementos. Como principal desventaja se encuentra que al ser una biblioteca reciente no cuenta con una documentación del todo completa, además que cada versión renueva muchos elementos por lo que tutoriales, guías o soluciones a problemas pasados cambian completamente de un año a otro.

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[32]:

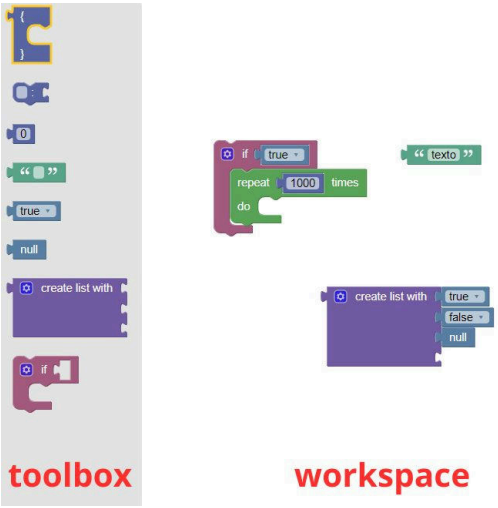


Figura 18: estructura básica de blockly

Un proyecto de blockly 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 programador 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 3 pasos desde el punto de vista del desarrollador[33]:

- 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 y JSON, aunque Google recomienda el uso de JSON.

Estos tres pasos se ven de la siguiente manera en código. Suponiendo la creación de un simple bloque con un cuadro de texto rellenable por el usuario:

Apariencia

```
export const blocks = Blockly.common.createBlockDefinitionsFromJsonArray([{
{
    "type": 'text',
    "message0": "%1",
    "args0": [
        {
            "type": "field_input",
            "name": "FIELDNAME",
            "text": "default text",
            "spellcheck": false
        }
    ],
}
}]);
```

Generación

```
import {javascriptGenerator} from 'blockly/javascript';
javascriptGenerator.forBlock = function(block) {
  var text_fieldname = block.getFieldValue('FIELDNAME');
  var code = '"' + text_fieldname + '"';
  return [code, Blockly.JavaScript.ORDER_ATOMIC];
};
```

Inclusión

```
export const toolbox = {
  'kind': 'flyoutToolbox',
  'contents': [
    {
      {
        'kind': 'block',
        'type': 'text'
      }
}]};
```

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.

La apariencia de blockly en nuestra aplicación es la siguiente:

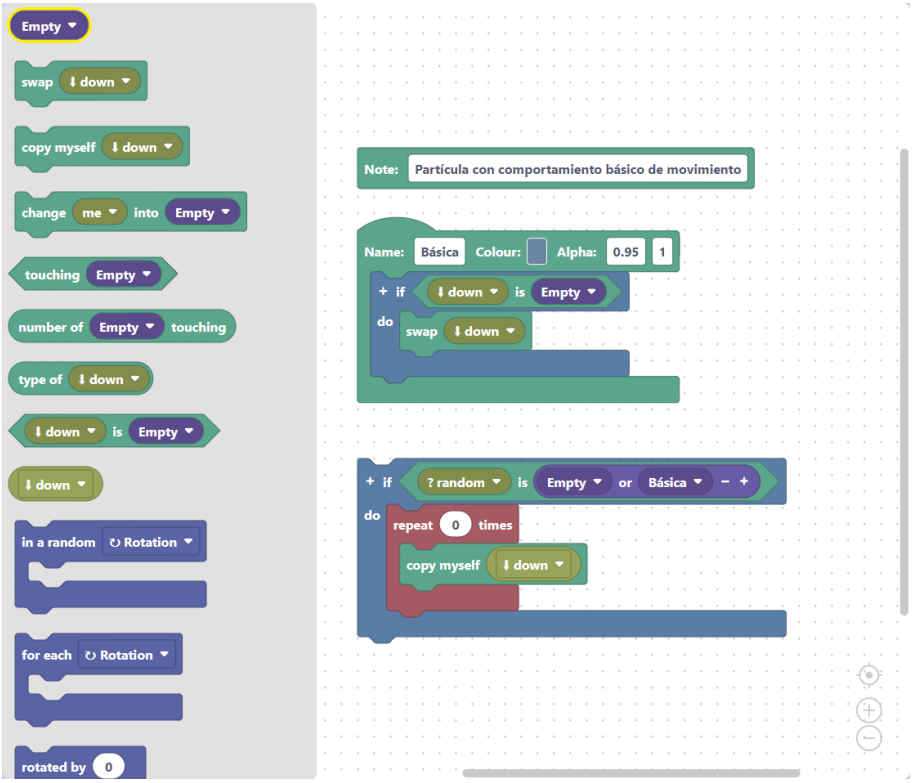


Figura 19: interfaz blockly pixel creator

la apariencia visual es igual a la apariencia general presentada anteriormente. Como elementos particulares, se encuentran 3 botones en la zona inferior derecha con el siguiente comportamiento ordenado de arriba a abajo: centrar la pantalla, aumentar el zoom, disminuir el zoom. Además, los bloques al moverlos se colapsan sobre una grilla definida por los puntos visibles en el workspace.

Cada partícula nueva crea un workspace nuevo e independiente del resto. Este workspace es creado con un bloque por defecto que define atributos básicos para la particula como el nombre, el color o el rango de transparencia aleatoria que puede tener cada instancia de partícula creada. Todo bloque que no se encuentre contenido dentro de este bloque no será procesado por el intérprete de JSON.

14 Simulador en GPU

Existe una distinción con respecto a las simulaciones anteriores en cuanto al procesado de partículas. En la introducción del capítulo anterior, Simulador CPU Sección 12.1, se declara que la implementación de los simuladores de arena difiere de los automatas celulares en que las partículas pueden observar los cambios que se han producido en otras partículas al momento, sin tener que esperar a que se procesen todas.

Por el funcionamiento y arquitectura de la GPU, cada hilo de ejecución, que procesará una partícula independiente, debe ser completamente ajeno a cambios de las partículas de su alrededor. Podría manejarse también haciendo un uso muy controlado de los hilos, bloques de hilos y de la memoria compartida, pero en cierta manera el establecer tanto control para el actualizado de partículas limitaría la fortaleza de la GPU, que es paralelizar lo máximo posible una tarea. Por lo tanto la aproximación que toma este simulador es de que cada celda sea capaz de determinar su estado en el siguiente frame con solo la información de sus vecinas, de la misma manera que cualquier autómata celular.

Esto a su vez, trae una serie de problemas, y es que a mayor sea la cantidad de tipos de partículas que existan en el mapa, mas variabilidad habrá a la hora de decidir el siguiente estado de la simulación. Por estos problemas, este proyecto solo dispone de la partícula de arena para probar su funcionamiento.

El procesamiento de partículas se realiza mediante un doble buffer, donde se lee la información de partículas de input, se determina el nuevo estado por partícula, se escribe en el buffer de output y se intercambian los buffers para volver a realizar la misma secuencia.

La lógica de una partícula de roca que se apilase verticalmente sería el siguiente:

- Si el hilo esta procesando una partícula vacía , comprueba si encima suyo hay una partícla de roca, en cuyo caso, sobreescribe la partícula actual para que sea una de roca.
- En caso de que esté observando una partícula de roca, solo hay dos opciones, o la partícula inmediatamente inferior es vacía y puede caer, en cuyo caso la posición actual se convierte en vacío, o la partícula inferior es roca y no puede caer.

Véase que la partícula no se preocupa acerca del comportamiento o estado mas allá de las casillas vecinas , ya que esto implicaría la necesidad de una ejecución secuencial. Este comportamiento provoca que se generen huecos a la hora de colocar partículas, ya que en el aire se van separando de la misma manera que se da en los simuladores de CPU, pero en este caso en se produce esta separación entre todas las lineas de partículas.

Para añadir otros movimientos, en concreto, moverse hacia abajo a la izquierda y abajo a la derecha para añadir comportamiento de arena, es necesario crear un compute shader diferente y ejecutarlos de manera separada, ya que si se intentase realizar todo el movimiento en una sola pasada, se generarían inconsistencias por obvios motivos. Una partícula, si bien no necesita conocer el estado inmediato de las vecinas, si necesita el inmediatamente anterior para poder determinar su próximo estado. Por lo que el movimiento de una partícula de arena comprueba, según la cantidad de movimiento por frame que tenga que realizar, si se puede mover hacia abajo, y hacia abajo a la izquiera o derecha y se repite el bucle hasta que se terminen los movimientos por realizar.

Esta implementación, a cambio de ser la más rápida en ejecución, no aporta flexibilidad al usuario, ya que el código de lógica de movimiento se ejecuta mediante compute shaders escritos en .GLSL, y requiere trabajo pensar en las interacciones entre partículas.

Se utilizó Vulkano, una librería hecha en Rust que actúa como wrapper de Vulkan 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, motor de videojuegos hecho en Rust para implementar mecánicas básicas como el bucle principal de juego, manejo de interfaces, procesamiento de input etc.

15 Comparación y pruebas

Una vez entendidos los automátas celulares 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.

15.1 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

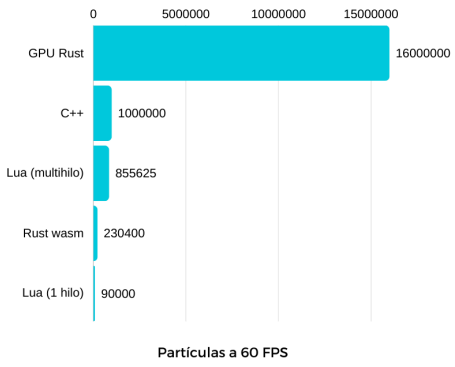
Equipo 2:
- CPU: Intel Core i7-10750H
- GPU: NVIDIA GeForce RTX 2060
- RAM: 16 GB
- Sistema operativo: Windows 11

Se han realizado las diversas pruebas:

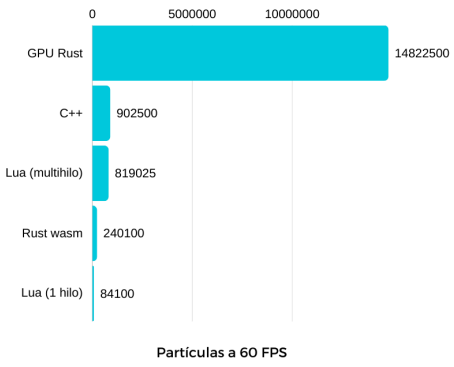
- Comparación entre todos los simuladores con el mismo tipo de partícula.
- Comparación entre los simuladores de CPU con una partícula demandante.

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

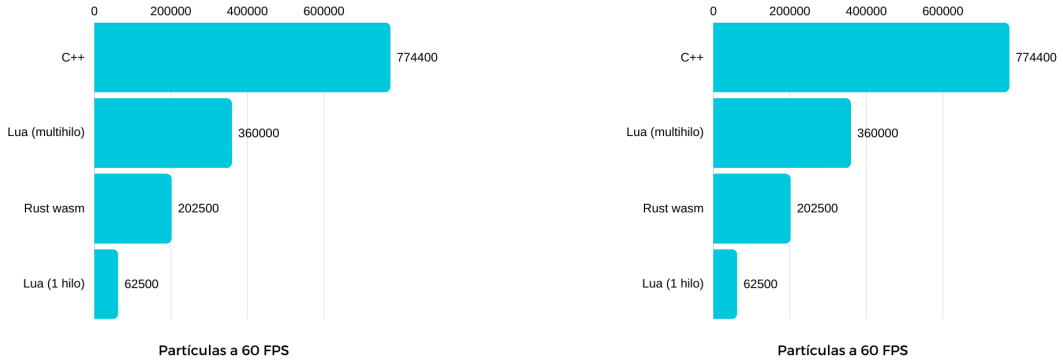


(b) Resultados segundo equipo

Figura 20: 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 20, 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 21: 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 20

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

15.2 Comparación de usabilidad

Para evaluar la usabilidad de los distintos sistemas se ha realizado una encuesta a un grupo de 8 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:

— Insertar parametros

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:

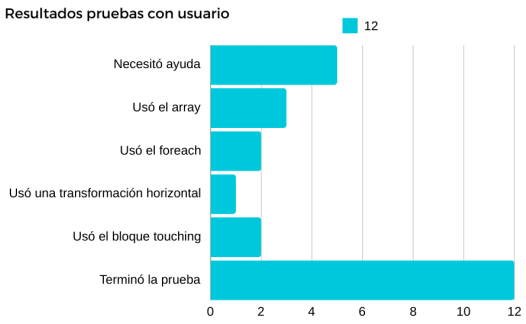


Figura 22: Resultados de las pruebas con usuarios

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.

16 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 nos ha permitido 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.

17 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.

18 Contribuciones

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

18.1 Nicolás Rosa Caballero

Según la normativa hay que sacar 4 páginas de esto, 2 cada uno pero a pesar de que consider que he hecho muchísimas cosas enumerarlas no ocupa tanto espacio. Además de que estoy poniendo solo lo que al final se ha quedado, hay cosas que al final se descartaron u otras que no pero que fueron problemáticas y llevaron mucho tiempo. Esta simple enumeración no refleja el esfuerzo que ha llevado cada una de estas tareas.

Trato de hablar impersonal en este apartado pero dado que es un apartado de contribuciones no sé si sería posible hablar en primera persona.

Existen 4 aportaciones principales de Nicolás al proyecto:

- Simulador en C++

Se realizó la maquetación inicial del proyecto, así como la implementación de OpenGL, GLFW y IMGUI. Además, se encargó de la implementación de la lógica inicial de la simulación de arena y la interacción con el usuario. Se implementaron las particula básicas para este modelo: Arena, Agua, Ácido, Roca, Aire y Gas. Se creó e implementó parte del sistema de interacciones entre partículas. Se realizaron las 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

Esta implementación fue completamente realizada por Nicolás. Para ello se implementó la lógica de la simulación de arena, el API de interacción de partículas, la interfaz con IMGUI, un sistema de entidades para manejar distintos objetos del programa y el sistema multithreading. Para el sistema multithreading se diseñó un sistema de comunicación y sincronización entre threads mediante canales. También se implementó un sistema de procesamiento basado en doble buffer para lidiar con las condiciones de carreras y asegurar la consistencia de los datos. Finalmente se elaboró un documento introductorio a Lua que describe el API del sistema para que los usuarios tuvieran una referencia de como extender el simulador.

- Simulador en Rust con Macroquad

Al igual que en la implementación anterior, todo el código fue escrito por Nicolás. Sin embargo, aunque la base inicial del proyecto fue realizada en solitario, el desarrollo de los plugins web fue asistido por Jonathan. Nicolás diseño la arquitectura del proyecto basada en plugins y el soporte para poder serializar y deserializar datos de un fichero JSON. También implementó la lógica que interpreta dichos datos, así como el API que mediante instrucciones se representa en Blockly como bloques. Este API puede ser usado también por partículas programadas en Rust compiladas a DLLs.

- Web Vue3 envoltorio del ejecutable de Rust

La web fue realizada en su mayoría por Nicolás, siendo la mayor excepción Blockly, en lo cual solo se ayudó a su integración en el sitio y comunicación con el binario mediante código pegamento de JavaScript. El diseño de la web, botones de partículas, sonidos, menú de ayuda, gestos,

sistema de guardado y cargado de plugins, integración continúa y logo fueron implementados por Nicolás.

- Memoria

No sé si este apartado es necesario

Nicolás creó la estructura de la memoria, la portada, el índice, la sección de autómatas celulares (mas no la de simuladores de arena), la sección de Plugins y Lenguajes de scripting, la sección de CPU y multithreading, la sección de Simualdor en CPU y parte de las secciones de comparación y conclusiones.

18.2 Jonathan Andrade Gordillo

Bibliografía

1. Li, T. M. *Cellular automata*. (Nova Science Pub Incorporated, 2011).
2. Adamatzky, A. *Game of Life Cellular Automata*. (2010). doi:10.1007/978-1-84996-217-9
3. Hadeler, K.-P. & Müller, J. *Cellular Automata: Analysis and Applications*. (Springer, 2017).
4. Schwartz, J. T., Von Neumann, J. & Burks, A. W. Theory of Self-Reproducing Automata. *Mathematics of computation* **21,** 745-746 (1967)
5. Conway's Game of Life: Scientific American, October 1970. https://www.ibiblio.org/lifepatterns/october1970.html
6. Wolfram, S. *A New Kind of Science*. (2002).
7. Weisstein, Eric W. Langton's Ant – from Wolfram MathWorld. https://mathworld.wolfram.com/LangtonsAnt.html
8. Wikipedia. Computational fluid dynamics. https://en.wikipedia.org/wiki/Computational_fluid_dynamics
9. Wikipedia. Particle System. https://en.wikipedia.org/wiki/Particle_system
10. Wikipedia. Sand Simulator. https://en.wikipedia.org/wiki/Falling-sand_game
11. Open System for Earthquake Engineering Simulation. https://opensees.berkeley.edu/
12. SimVascular. https://simvascular.github.io/documentation/flowsolver.html
13. Archive, W. Falling Sand Game. https://web.archive.org/web/20090423105358/http://fallingsandgame.com/overview/index.html
14. Toy, P. Powder Toy. https://powdertoy.co.uk/
15. Bittker, M. Sandspiel. https://maxbittker.com/making-sandspiel
16. Bittker, M. Sandspiel Club. https://sandspiel.club/
17. Google. Blockly. https://developers.google.com/blockly
18. Sutherland, I. E. A man-machine graphical communication system. (1963).
19. Stallings, W. *Computer Organization and Architecture Designing for Performance*.
20. Peddie, J. *The History of the GPU - Steps to Invention*. vol. 1 (2023).
21. Aamodt, T. M., Fung, W. W. L. & Rogers, T. G. *General-Purpose Graphics Processor Architectures*. (Springer Nature, 2022).
22. Krüger, J. & Westermann, R. Linear algebra operators for GPU implementation of numerical algorithms. (2003).
23. Patterson, D. A. & Hennessy, J. L. *Computer Organization and Design*. (Morgan Kaufmann, 2013).
24. Möller, T., Haines, E. & Hoffman, N. *Real-time renderiNg*. (A K PETERS, 2018).
25. Compute Shaders Introduction. https://learnopengl.com/Guest-Articles/2022/Compute-Shaders/Introduction

26. Bustio-Martínez, L., Peña, Y. C. & Bustamante, I. T. *Arquitectura basada en plugins para el desarrollo de software científico*. (2013).

27. SPARC Internal White Paper. Understanding the Application Binary Interface. https://link.springer.com/content/pdf/10.1007/978-1-4612-3192-9_25.pdf

28. «Assemblies in .NET - .NET», Microsoft Learn, 15 de marzo de 2023. https://learn.microsoft.com/en-us/dotnet/standard/assembly/.

29. «Creating a resource-only DLL», Microsoft Learn, 3 de agosto de 2021. https://learn.microsoft.com/en-us/cpp/build/creating-a-resource-only-dll.

30. Levine, J. R. *Linkers and Loaders*. (Morgan Kaufmann, 1999).

31. «The Rustonomicon: Alternative representations». https://doc.rust-lang.org/nomicon/other-reprs.html

32. Blockly Visual. https://developers.google.com/blockly/guides/get-started/workspace-anatomy

33. Blockly Block Creation. https://developers.google.com/blockly/guides/create-custom-blocks/define-blocks