Autómatas en dos dimensiones tipo Life

Sistemas Complejos

Prof: Genaro Juárez Martínez

Luis Eduardo Valle Martínez

24 de Abril del 2022

1 Introducción

Este reporte técnico tiene como objetivo explicar y documentar la solución propuesta mediante el diseño y desarrollo de un simulador de espacios de evolución de Autómatas Celulares en 2 dimensiones mediante reglas de evolución definidas, siendo el referente base el trabajo desarrollado por el matemático británico John Horton Conway, el Juego de la Vida de Conway.

Este juego se trata de un juego tipo jugador zero, donde la evolución del sistema está determinado por un estado inicial y no requiriendo otra entrada para observar su evolución.

El universo del Juego de la Vida es teóricamente un espacio infinito, una rejilla ortogonal bidimensional de células cuadradas, de las cuales pueden tomar 2 únicos valores(Viva o Muerta), correspondiento al alfabeto $\Sigma = 0, 1$. Cada una de las células interactua con sus 8 vecinos que la rodea, llamada vecindad de Moore(ventana del espacio de evolución de tamaño 3x3 donde la célula ancla ubicada en el centro de la ventana interactua con las células que la rodean para definir el estado siguiente en una nueva generación).

Por términos prácticos, el universo de una simulación del juego de la vida no puede ser inifinito, por lo que se suele indicar el tamaño de espacio de evolución antes de iniciar la ejecución del programa con un valor asignado a una constante de configuración.

Las evoluciones del universo en la simulación dependerá en cada célula por su vecindad de Moore, rigiéndose por una regla que indica según el número de vecinos adyacentes vivos el nuevo estado que toma la célula. Para la regla del Juego de la Vida las condiciones son las siguientes:

1. Cualquier célula viva con 2 o 3 vecinos vivos sobrevive.
2. Cualquier célula muerta con exactamente 3 vecinos se converte en una célula viva(Nacimiento).
3. Cualquier célula con más de 3 vecinos vivos muere por sobrepoblación
4. Cualquier célula con menos de 2 vecinos vivos mere por subpoblación

Por naturaleza esta implementación de reglas se realiza simultaneamente en cada célula obteníendose los nuevos valores de las células en el mismo instante. Cada generación es una función pura de las precedentes. Esta regla es implementada repetidamente para la creación de futuras generaciones.

El desarrollo se realizó utilizando el lenguaje de programación Python con el uso de la biblioteca PyGame como principal bilioteca gráfica, Numpy para el manejo eficiente de estructuras como *arrays* para los espacios de evolución, Matplotlib para la graficación de los análisis estadísticos y finalmente Numba como traductor de funciones Python a funciones paralelizables con aceleración por hardware gracias al controlador CUDA de NVIDIA para sus GPUs.

La simulación ofrece su funcionalidad mediante el control enteramente desde la interfaz, la cual se diseño considerando una fácil e intuitiva interacción para los usuarios, el mayor aprovechamiento del espacio y estructura de los componentes que la compone y finalmente un diseño agradable y consistente que pueda ser común con otras interfaces populares acelerando la adaptación de un nuevo usuario con la aplicación.

En secciones posteriores se realizará el desglose del programa en sus dependencias requeridas para se ejecución, requsitos funcionales del programa, interfaz y su conformación, elementos gráficos, su funcionalidad y operación, la implementación de optimización mediante aceleración por hardware, características del espacio de evolución y exposición de todas las funcionalidades de este, ejemplos de la simulación corriendo a diferentes tamaño de evolución, graficación de las medidas de análisis estadístico por generaciones, y finalmente el código fuente del programa completo.

2 Programa

2.1 Dependencias

A continuación se listan las dependencias de bibliotecas, programas y hardware requeridos para la ejecución del programa.

El programa fue desarrollado utilizando el lenguaje de programación Python por lo que se requiere minimamente la instalación de Python en culquiera de sus versiones 3, siendo desarrollado en la versión 3.9.7.

Las bibliotecas listadas a continuación pueden ser instaladas mediante programas como *pip*, o directamente en un ambiente virtual como *pipenv* o *anaconda*:

- PyGame 2.1.2
- Numpy 1.21.5
- Matplotlib 3.5.1
- Numba 0.55.1

PyGame es una biblioteca utilizada para el desarrollo de sencillos juegos 2D en python, incluyendo algunas herramientas gráficas para la impresión en pantalla de diferentes forma geométricas, imágenes y Sprites. Incluye métodos usados para la detección de colisiones entre objetos, impresión de áreas en pantalla y otros de configuración como el número de veces de refresco de pantalla en cuadros por segundo FPS(*Frames Per Second*). Incluye así también métodos para la fácil implementación de sonido y archivos de audio, sin embargo este tipo de funciones no se utilizan en el desarrollo de este programa.

Numpy es un biblioteca para el manejo de vectores y matrices, incluyendo un amplio conjunto de recursos en funciones matemáticas en el dominio del algebra lineal, transformada de fourier y matrices. Principalmente el uso de esta biblioteca en el programa se encuentra en el uso de *arrays* como estructuras para el manejo de espacios y otros recursos auxiliares que permiten de forma sencilla y eficiente las operaciones durante la evolución del espacio bidimensional de los autómatas. Importante señalar que los *arrays* y algunas operaciones de *numpy*, son las estructuras básicas y funciones matemáticas válidas en *Kernels* y funciones numba utilizadas para el procesamiento con CUDA.

Matplotlib es una biblioteca utilizada para la creación de visualizaciones estáticas, animadas e interactivas en Python. Esta biblioteca provee funciones que permiten la graficación de las diferentes medidas de complejidad.

Numba es una biblioteca de Python que traduce funciones a código máquina optimizando el tiempo de corrida utilizando la biblioteca de compilación estándar en la industria LLVM. Algoritmos numéricos compilados por Numba en Python pueden alcanzar velocidades de C o Fortran, a través de la paralelización de algoritmos ofreciendo un rango de opciones utilizando CPUs y GPUs. En el programa se utiliza una alternativa de aceleración de hardware al tener soporte para CUDA de Nvidia y ROCm de AMD para sus tarjetas gráficas. Mediante la creación de *Kernels*(Función ejecutada en un número de bloques con N números de hilos de ejecución para cada bloque), esto resulta conveniente pues el tiempo de procesamiento de los *array* se mantiene relativamente constante pues se ocupan NxM hilos siendo NxM el tamaño del *array*.

La totalidad del proceso de desarrollo y pruebas del funcionamiento de la simulación se realizó utilizando un SO llamado Pop!_OS en su versión 21.10, sistema basado en UNIX Linux derivado de la popular distribución Debian. Se utilizó un ambiente virtual para la ejecución de la simulación, con la instalación de dependencias únicamente en este ambiente y no todo el SO con la herramienta *pipenv* de Python.

La ejecución del programa en otro sistema operativo diferente a una distribución Linux no debería ser impedida si se cuenta con la instalación de todas las dependencias, ya sea en un ambiente virtual o directamente en el ambiente de trabajo del usuario. El único inconveniente que pudiera surgir aún con las depedencias instaladas es el uso de la aceleración por hardware, si el equipo que se utiliza no cuenta con una tarjeta gráfica dedicada de NVIDIA o AMD, es necesario modificar la constante de configuración **GPU_ENHANCEMENT** a *False* en la línea 28 del archivo *main.py*.

2.1.1 Hardware

A raíz de la optimización del programa para su ejecución con la ayuda de aceleración por GPU se requiere contar evidentemente con una GPU, sin importar sea de NVIDA o AMD.

Requiriendo si por su parte, la instalación de los controladores CUDA para NVIDIA y ROCm para AMD en el sistema.

La aceleración por GPU refiere a la paralelización de los algoritmos realizados principalmente sobre estructuras tipo *arrays*, dando la posibilidad de ejecutar una función para cada elemento del array en un núcleo propio y lo que permite se acelere mucho el tiempo de corrida de un algoritmo que en un CPU correría secuencialmente mientras que con la GPU tomaría un tiempo comparable a la ejecución de la función para 1 de los elementos del *array* con la típica ejecución mediante CPU.

2.2 Requerimientos

El programa consiste en un simulador gráfico de un espacio de evoución para una cierta configuración de Autómatas Celulares en 2 dimensiones, basado en el "Juego de la vida" de Conwell. Precisamente la evolución se realiza siguiendo una tupla de números que corresponde a una regla, dependiendo del valor de estos 4 números y el número de vecinos en una vecindad de Moore, en la siguiente evolución la célula ancla puede Vivir, Morir o Nacer, dependiendo el estado nuevo también del estado actual de la célula(Viva o Muerta).

Para facilitar el uso del simulador y precisamente poder observar la evolución de una manera intuitiva en forma de animación, el programa requirió de una implementación gráfica, aprovechandose esta cualidad para incluir botones o diferentes componenetes que permitan al usuario configurar los parámetros y condiciones en el que tendrá lugar el proceso de evolución para la configuración del espacio.

A continuación se numeran los requerimientos funcionales con las que el programa debe cumplir:

1. Evaluación de espacios de 300x300, 500x500 y máximo de 1000x1000 células
2. Animación en 2 dimensiones
3. Poder cambiar los colores de los estados
4. Inicialización del espacio de evoluciones con diferentes densidades
5. Edición del espacio de evoluciones para dibujar configuraciones particulares
6. Salvado y levantado de archivos con configuraciones en específico
7. Graficación de:
    - Densidad(número células vivas) por generación
    - Logaritmo base 10 de la Densidad por generación
    - Entropía
8. Especificación de las reglas de evolución en notación: $R(S_{min}, S_{max}, B_{min}, B_{max})$

2.3 Interfaz

Como se mencionó en la sección de **Dependencias**, la principal biblioteca para los elementos gráficos del simulador fue *PyGame*.

Aún con las herramientas y funciones que provee esta biblioteca, se considera una biblioteca de nivel bajo, facilitando tan solo herramientas básicas para la construcción completa de un videojuego, lo que ya en un principio presenta un conjunto de retos que se exponen posteriormente, así como la solución con la que se abordaron estas limitaciones.

Importante es tomar en cuenta que como una biblioteca enfocada a videojuegos, no se proveé prácticamente facilidad alguna para la creación de interfaces gráficas con los típicos elementos gráficos como botones, sliders, etc. Se resalta esto pues existen bibliotecas sencillas en Python como *TKinter* o incluso *Frameworks* ya más complejos como *PySide 6*, que proveen herramientas avanzadas para la creación sencilla de interfaces gráficas, con la posibilidad de implementar diseños modernos con funcionalidades intuitivas sin demasiada dificultad. El caso de *PyGame* es lo contrario, todos los elementos gráficos e incluso de estructura, fueron implementados desde 0 construyendose con formas básicas como rectángulos, círculos, iconos e inclusive el posicionamiento de los elementos se realiza a nivel coordenadas de pixel dentro la ventana.

Ya creados estos elementos básicos gráficos para conformar una interfaz, en conjunto de los elementos de organización y posición, se muestra en la figura 2.3 la interfaz final del simulador.

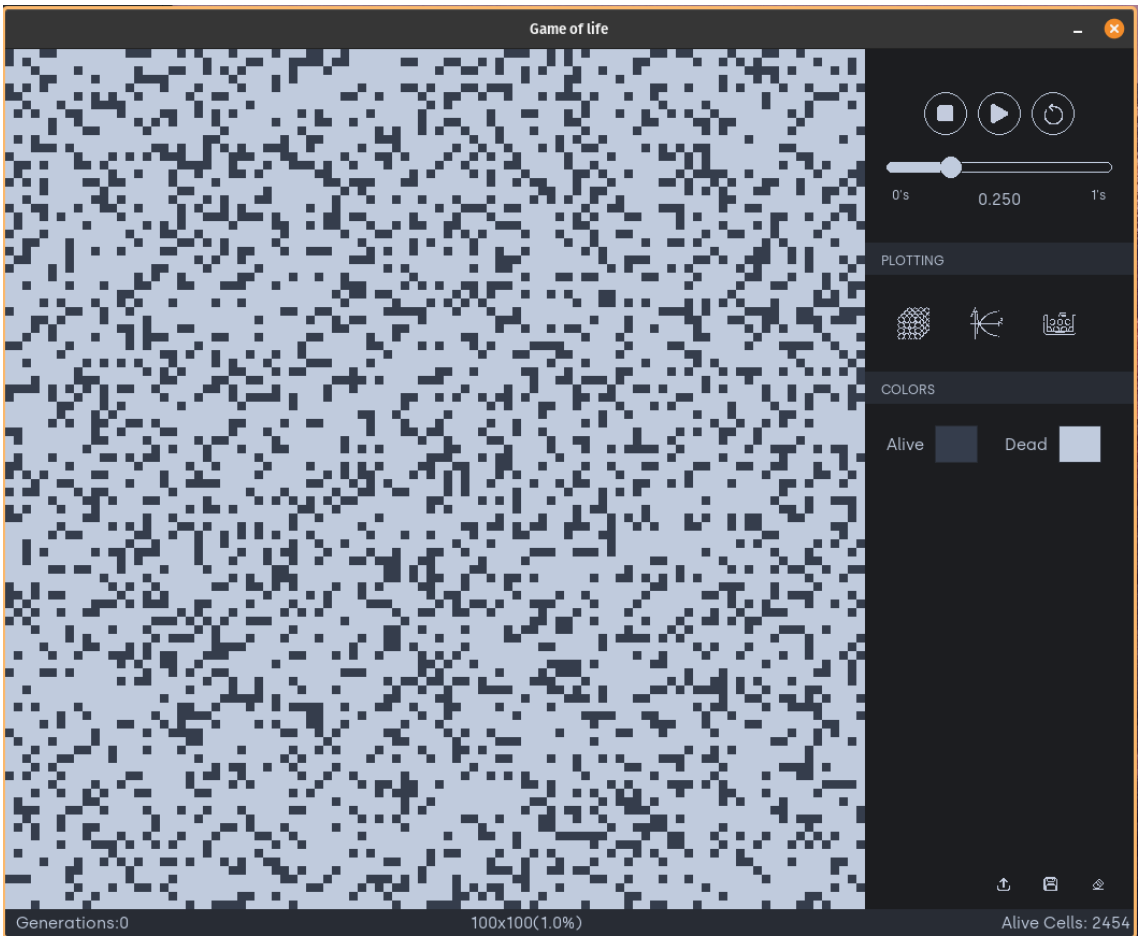

Figure 1: Captura de pantalla de la interfaz gráfica del simulador al ejecutarse

2.3.1 Componentes Gráficos

Todos los componenetes gráficos que se describen pertenecen al módulo de la aplicación nombrado *GraphicalComponents* y que puede verse su implementación Python en la sección de **Código Fuente**.

Botón Los botones son componentes gráficos básicos en cualquier GUI, y no es excepción en el programa de simulación pues se utilizan un total de 9 botones y que permiten desempeñar diferentes acciones.

La primer triada de botones son los utilizados para Pausar, Correr y Cargar una nueva configuración de la simulación.

Ubicados en la parte superior derecha en la sección lateral, son probablemente los botones más importantes pues ejecutan las funciones básicas de evolución en la simulación.

Figure 2: Botones que dictan el flujo de las evoluciones del espacio 2D

Siguiendo en importancia ubicados en la parte inferior de la sección lateral se ubican los 3 botones de acciones relacionadas al espacio gráfico de evolución. El primero de ellos representado por un ícono de una flecha es el botón *Charge* y permite cargar una configuración en un archivo CSV. Le sigue el botón con ícono de disquet *Load*, que permite guardar la actual configuración del espacio de evolución. Finalmente se tiene un ícono de goma de borrar *Clear*, que limpia el espacio de evolución colocando el espacio con todas las células muertas.

Figure 3: Botones inferiores para la configuración del estado de evolución

Finalmente ubicadas en la sección de *Plotting*, se tienen 3 botones que se encargan de la graficación. El primero de ellos con un ícono de cubo corresponde a la gráfica de Densidad para las generaciones corridas. Le sigue un botón con ícono de una gráfica logarítmica, y que precisamente grafica el logaritmo de las densidades. Finalmente se tiene el ícono con unas partículas de un gas para la graficación de la Entropía.

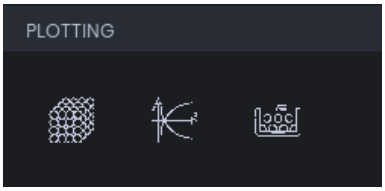

Figure 4: Botones con íconos para la graficación de la complejidad de los estados evolucionados

Texto Se utilizan algunos textos para indicar el estado actual del espacio de evolución, siendo los 3 más importantes aquellos impresos en la barra inferior, y que respectivamente indican el número de generaciones que han transcurrido, el tamaño del espacio y entre paréntesis la escala de zoom(Funcionalidad no disponible con la versión de la simulación en el tiempo en el que se redacta el reporte), y finalmente el número de células en el espacio que se encuentran vivas.

Figure 5: Textos ubicados en la barra inferior. Con el proposito de mostrarlos los 3 la imagen se ha recortado para ubicarlos juntos, sin embargo en la interfaz se muestran en las esquinas y el centro de la ventana en la parte inferior

Slider Un slider es un componente gráfico algo más complejo que va a estar formado por una combinación de los 2 elementos anteriores así como un par de rectángulos y círculos que le dan forma al cuerpo de la ranura por la que el botón se desliza.

La función de este slider es la elección del complemento de la densidad, y que es aplicado cuando se da al botón Reset que carga una nueva configuración de espacio de evolución con un número aleatorio de células vivas que se rigen por el complemento de la probabilidad que se escoge con el slider.

Figure 6: Slider para elegir la probabilidad de las células muertas cuando se realiza un Reset

Input El input es un elemento típicamente utilizado en formularios para el ingreso de texto corto. El input cuenta con una *Label* etiqueta que sirve al usuario para indicar el tipo de respuesta que se debe colocar ahí, asi como del cuerpo del input y el texto por supuesto.

Un uso poco convecional pero conveniente por su previo desarrollo e implementación en el programa de ECA en 1 dimensión, un par de inputs se utilizan como una clase de botones que permiten el cambio de los colores para las células vivas o muertas.

Se opto por utilizarse estos elementos por el texto de etiqueta que por defecto se coloca en el input, colocándose un texto vació e ignorando eventos de tecla que los pueda modificar, siendo únicamente el click el que desencadene la función para el cambio de color.

Los colores disponibles pueden consultarse en el módulo *Constants* en la sección **Código Fuente**.

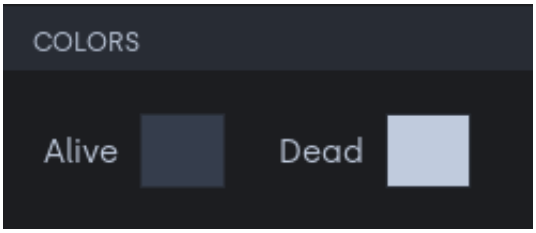

Figure 7: Par de inputs ubicados en la sección *Colors* y que permiten modificar el color actual de las células vivas y muertas

Células Gráficas Este componenete gráfico es el único que no se encuentra en el módulo *GraphicalComponents* pues se encuentra declarado en el módulo *Graphics*, esto se debe a que a diferencia de los elementos anteriores las células no son componentes comúnes en una biblioteca GUI de propósito general, siendo particulares de esta aplicación de simulación.

Las células gráficas son en escencia un *sprite* en forma de rectángulo que cambiará su color en los diferentes casos en función de si esa célula se encuentra viva o muerta:

- Se presiona o pasa encima el cursor mientras se mantiene presionado el click para cambiar el estado de la célula y por lo tanto su color.
- Durante el proceso de evolución la célula cambia su estado
- De forma general todas las células cambian a muertas cuando se da un *Clear*.
- Dependiendo de la configuración inicial aleatoria algunas células pueden cambiar su color pues se han definido como células vivas.

Figure 8: Fragmento de células gráficas en el espacio de evolución

Section Componente con representación gráfica en forma de una franja que contiene un encabezado con el título de la sección.

Este recurso se utiliza para realizar la separación visual de los botones de graficación y los colores de las células, pero también se utiliza como referencia para colocar los elementos gráficos clasificados dentro de esa sección.

2.3.2 Layouts

Como se explicó, el posicionamiento de los elementos en la ventana se realiza a nivel coordenadas de pixel, por lo que para algunos casos como la colocación de las células gráficas o las secciones y barras laterales e inferior, se utilizan componentes de *layout* u organización, que pueden o no tener alguna representación gráfica.

Estos objetos pueden encontrarse en el módulo *Layouts*

SideBar Componente *singleton*, instanciado 1 sola vez en la ventana, que tiene representación gráfica como un rectángulo en la parte lateral derecha de la ventana.

Semánticamente separa a la sección del espacio de evolución para albergar en su interior los diferentes componentes que controlan la evolución del simulador.

BottomBar Componente *singleton*, instanciado 1 sola vez en la ventana, que tiene representación gráfica como un rectángulo en la parte inferior de la ventana.

Semánticamente separa a la sección del espacio de evolución y la barra lateral, para albergar en su interior los textos que indican el estado de la evolución así como el tamaño y zoom.

Grid Componente sin representación gráfica que fue desarrollado para mimetizar el comportamiento de este tipo de organización gráfica en forma tabular.

Este objeto se encarga de calcular el tamaño de sus elementos según las medidas proporcionadas para ocupar todo el grid, y los paddings entre cada elemento.

Se le pasa una lista de los Rectángulos de los elementos gráficos para que según la disposición calculada automáticamente se coloquen los elementos en su respectiva posición, conformando finalmente el grid completo con las dimensiones y número de filas y columnas indicadas.

2.4 Aceleración por GPU

En secciones anteriores se ha explicado el uso de la biblioteca *Numba* como recurso para la optimización del tiempo de ejecución mediante la paralelización de los algoritmos relacionados con procesos numéricos en *arrays* al utilizarse el hardware de una tarjeta gráfica, aliviando un poco así también la carga del CPU y permitiendo el cálculo completo de proceso en un teórico ciclo de reloj.

Evidentemente no todos los sistemas donde es posible correr este algoritmo contarán con el hardware necesario para aprovechar esta aceleración, por lo que como una constante de configuración en el archivo *main.py*, se permite cambiar el valor en booleano para la activación manual de esta optimización: *GPU_ENHANCEMENT = True*.

2.4.1 Manejo de memoria

Cuando se trabaja con una GPU para el procesamiento númerico de *arrays*, se le denomina a esta *device* y al CPU *host*.

Un proceso que consume relativamente mucho tiempo cuando se maneja con funciones CUDA o funciones *Kernels* que operan en un dispositivo(GPU), es el traslado de los datos(*arrays*) entre el *host* y el *device*, y viceversa. Por esta razón existe un conjunto de funciones que permiten de antemano alojar en la memoria del dispositivo *arrays* que más tarde pueden ser accedidos por los hilos de todos los bloques.

Existen también otros tipos de reservación de memoria, donde unicamente los hilos de un bloque tienen accesso a ese *array* o para datos constantes, sin embargo este tipo de memorias no son utilizadas en la simulación de forma que solo se maneja la memoria general.

Las funciones utilizadas para reservar la memoria retornan un objeto que puede ser pasado como argumento en funciones CUDA o Kernel que serán utilizadas por estas como referencias para consultar el respectivo *array* en memoria de la GPU directamente, ahorrándose el tiempo requerido para copiar todos los *array* del *host* al dispositivo, ejecutar el algoritmo y realizar la copia de vuelta de todos los *array* a la memoria del CPU.

Esto implica entonces que los espacios de evolución actualizados que mantienen los valores reales actuales de las células en la generación actual, se encuentren en la memoria del GPU, requiriendo el uso de otra función que realiza una copia de los *array* en el dispositivo al *host*, sin embargo para cada función se requiere a lo más el recuperar 1 o 2 *array* para operar otro algoritmo en el CPU, y siendo en ocasiones ningúno el resultado requerido a copiar de vuelta, ahorrando un lapso importante de tiempo que disminuye el tiempo de respuesta.

Al mantenerse en memoria del *device* los espacios de evolución actualizados y otros *array* utilizados en diferentes funciones, se implementan algunas funciones *Kernel* que solo se utilizan para actualizar estos *array* y no para la ejecución de una algoritmo paralelizable. Aún con este inconveniente, el tiempo de ejecución es considerablemente más rápido con la aceleración con GPU que el procesamiento único del CPU.

2.4.2 Kernels

Igualmente se ha explicado anteriormente los *Kernels* los cuales permiten ejecutar funciones en un número de bloques con N hilos cada uno, por lo que de forma natural se describe la ejecución de operaciones en cada elemento del arreglo en un hilo independiente, consiguiendo un paralelismo real y no simple concurrencia con una teórica paralelización como se logra con diferentes hilos en el CPU.

El uso de *arrays* como estructura para almacenar el espacio de evolución y otros recursos, no solo es útil por la eficiencia computacional y sencillez de uso cuando se usa con procesamiento de CPU, pero es doblemente util cuando de utiliza la aceleración por GPU pues es la única y más utilizada estructura de operación en los *Kernels*.

Afortunadamente para los términos de la simulación, muchos de los procesos costosos computacionalmente pueden realizarse de forma paralela permitiendo la optimización de estos cálculos que de otra manera en un procesamiento secuencial de la CPU tendrían una complejidad cuadrada $O(n^2)$.

Se describen a continuación los *Kernels* implementados y que optimizan el tiempo de ejecución de la simulación cuando se activa la mejora por GPU(todas estas funciones pertenecen al módulo *CUDACellularAutomaton* y puede encontrarse en la sección de **Código Fuente**):

cuda_define_cell_status Función CUDA utilizada como función de apoyo durante el cálculo de una nueva generación. Este algoritmo en función de la regla que se está evaluando, el número de vecinos vivos en una vecindad y el valor de la célula ancla que se evalua, retorna el nuevo estado de la célula ancla indicándolo como 1 cuando vive o nace, y 0 cuando esta muere.

cuda_count_neighbours Al igual que la función anterior, se trata de un algoritmo de apoyo al cálculo de una neuva generación, retornando únicamente el número de células vivas identificadas en una ventana que corresponde a la vecindad de Moore de una célula.

cuda_next_generation Función *Kernel* utilizada para calcular el nuevo valor de una célula según su número de vecinos vivos en su vecindad de Moore y la regla con la que evoluciona el espacio. A diferencia de un algoritmo de procesamiento secuencial en CPU que requeriría de un par de bucles for para recorrer cada célula en el *array*, en este *Kernel* se obtiene la posición del elemento con el id del bloque(fila) y el id del hilo dentro del bloque(columna), implicando que el código y el procesamiento posterior se realice considerando el único elemento que localizan en conjunto este par de id's.

cuda_update_results Función *Kernel* que actualiza los resultados de una nueva evolución para ser almacenados en un array auxiliar que almacena el valor de las células en el espacio de evolución de la generación actual.

cuda_shannons_probability Función *Kernel* utilizada para el cálculo de la frecuencia de las vecindades para la entropía de Shannon. Con el mismo comportamiento que opera sobre un solo elemento del espacio de evolución, se identifica el tipo de vecindad utilizando una matriz de conversión que no es más que una matriz con las potencias de 2 en orden, de esta forma la vecindad de Moore actua como una máscara que después de sumarse los elementos resultantes de la multiplicación con la matriz de conversión se obtiene el número de la vecindad, la cual se utiliza para incrementar la frecuencia de ese número de vecindad en específico. Es importante mencionar que este método únicamente calcula las frecuencias de las vecindades, por lo que no retorna el valor de la entropía, calculándose ya en un proceso de CPU con la aplicación directa de la ecuación de la entropía, sin embargo parte del proceso más costoso ya se optimizó mediante esta paralelización.

cuda_change_cell Función *Kernel* utilizada para modificar el estado de 1 célula específica en el *array* localizado en la memoria del dispositivo. Esta función se utiliza comunmente cuando se cambia el estado de una célula gráficamente al dar click en ella.

cuda_clear_space Función *Kernel* que asigna a 0 los valores del par de arreglos que se almacenan el espacio actual y el de evolución alojados en la memoria del dispositivo.

2.5 Espacio de Evolución

El espacio de evolución dentro el programa se divide realmente en 2. El primero de ellos y el más evidente es el espacio gráfico, siendo visible mediante la configuración en *grid* de las células gráficas como el componente principal que ocupa el mayor porcentaje de espacio en la ventana, teniendo una relación directa este espacio con funciones meramente gráficas como el cambio de estado de una o un conjunto de células al hacer click sobre estas o arrastrar el cursor mientras se mantiene presionado, permitiendo realizar evoluciones sobre el espacio de manera particular con configuraciones específicas.

Aún con esta, se define un segundo espacio en el dominio lógico sobre el cual se realizan todas las operaciones de evolución, análisis estadístico, etc. Este espacio es representado mediante una estructura *array* en 3 dimensiones, donde las primeras 2 corresponderán al número de células y en la tercer dimensión se compone de 2 columnas, la primera de ellas se utiliza para almacenar el valor actual de la última evolución calculada, mientras que la segunda columna será el espacio donde se almacenen los estados de la nueva generación. Con esta técnica se evita calcular valores erroneos de las células al considerar vecindades de Moore afectadas por las más recientes modificaciones de los valores de las células por el mismo proceso de evolución.

Es evidente que ambos dominios se mantienen relacionados para asegurar la congruencia entre la evidencia gráfica del espacio y el espacio lógico, haciendo posible que una modificación en una célula desde la GUI permita que en un proceso de evolución se considere el valor nuevo asignado a la célula en la posición del grid. De esta relación ambas clases *GameGraphics* y *CellularAutomaton* implementan métodos para el cambio de los estados de las células gráficas y lógicas, incluyendo incluso una referencia a la instancia de la otra clase la una de la otra.

2.5.1 Configuración del espacio de evolución

En el archivo Python nombrado como *main.py* se tienen un conjunto de constantes de configuración que pueden ser modificadas para cambiar los aspectos gráficos e incluso lógicos de la simulación, no siendo la excepción el espacio de evolución donde los valores de las siguientes constantes influyen en este:

- **GRID_SIZE_ELEMENTS**: Indica el número de células por columnas como así también el número de filas, por lo que el tamaño del espacio es el cuadrado de esta constante, $GRID\_SIZE\_ELEMENTS^2$
- **GRID_PADDING**: En el aspecto gráfico, esta constante se modifica para añadir espaciado entre célula y célula en el grid
- **GRID_SIZE**: Tamaño en pixeles del ancho y altura del grid. Es importante que este valor sea al menos igual al *GRID_SIZE_ELEMENTS* para tener células de 1px, de otra forma se da un error pues no se pueden crear células gráficas con tamaño menor al pixel.

2.5.2 Configuración aleatoria inicial

La configuración aleatoria inicial consiste en definir una configuración nueva en el espacio de evolución en dependencia de la probabilidad de densidad para que el nuevo estado de la célula sea viva. Esta probabilidad de densidad es configurable directamente utilizando el *slider* gráfico(Figura 6), iniciando en primera instancia inmediatamente después de correr la simulación en 50% para ambos estados.

Si se desea cargar una nueva configuración aleatoria inicial, ya sea con una nueva o la misma probabilidad, se puede dar click al botón de reset(Figura 2)

2.5.3 Configuraciones personalizadas

Mediante un click o directamente arrastrando el puntero sobre las células gráficas, se puede crear una configuración personalizada sobre el espacio de evolución. Lo que sucede al dar click o arrstrar sosteniendo click es que las células que colisionan con el puntero van a cambiar su estado actual al estado contrario(Viva → Muerta o Muerta → Viva).

Un ejemplo de la conformación personalizada de configuraciones en un espacio de evolución se observa en la Figura 9

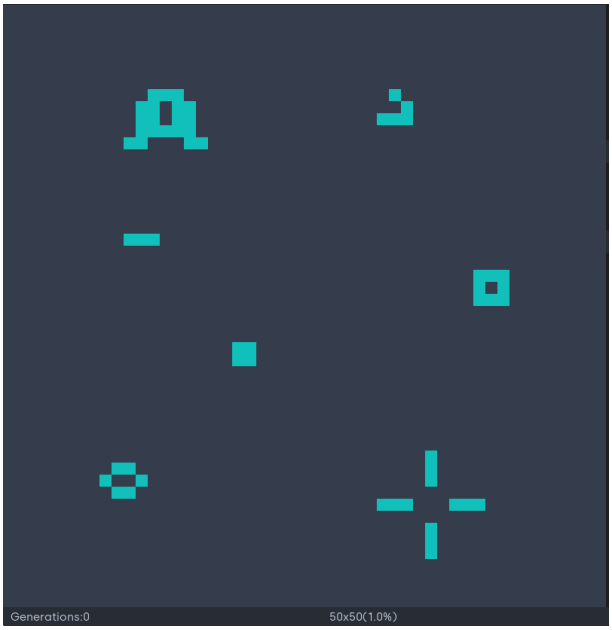

Figure 9: Ejemplo de una configuración personalizada en un espacio de 50x50 con algunas de las estructuras comunmente encontradas en espacios de evolución suficientemente grandes o mediante trabajos de investigación especializados

2.5.4 Limpiar, Guardar o Cargar espacio de evolución

Controlado por la triada de botones en la parte inferior derecha(Figura 3), permite a la simulación limpiar completamente el espacio de evolución cambiando el estado de todas las células a 'Muertas' en ambos espacios(gráfico y lógico) y colocando en 0's el valor de las generaciones y células vivas.

El par restante de botones son el complemento el uno del otro de una acción. La acción de guardar el espacio de evolución va a generar un nuevo archivo con extensión CSV guardándose en la carpeta *saves*. Este archivo nuevo representa en 0's y 1's el estado del espacio de evolución actual, razón por la cual si se cuenta con un archivo con estas características se puede cargar de vuelta en una simulación.

El proceso de cargado de un espacio guardado requiere que dentro de la carpeta *saves* se tenga el archivo con la configuración deseada renombrado a *upload.csv*, siendo imposible por el momento la elección específica de otro archivo e incluso generando un error si se acciona el botón de *Upload* sin existir este archivo en la carpeta. Este proceso permite cargar configuraciones de espacios de evolución iguales o más pequeños que el espacio actual, centrándose la configuración cargada con respecto a filas y columnas si esta es más pequeña(Figura 10).

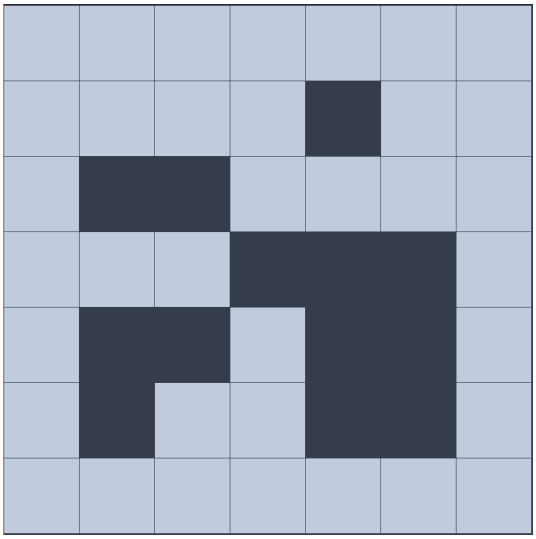

Figure 10: Ejemplo del centrado de una configuración de tamaño 5x5 cargada en un espacio de evolución 7x7

2.5.5 Cambio de colores

La simulación permite cambiar los colores de las células gráficas de ambos estados, contando para esto de un par de componentes gráficos(Figura 7) a los cuales al dar un click sobre estos cambiará su color, tanto el recuadro del input como el grid de las células gráficas que actualmente tengan el estado modificado, por el siguiente en la lista llamada *COLORS_LIST* en el módulo *Constant*. Esta lista no es más que un conjunto de 13 tuplas con 3 elementos cada una, representando cada tupla un color en formato RGB.

Por el momento el cambio de colores solo se logra de esta forma por lo que no es posible asignar un color específico no identificado en el módulo, sin embargo esta cantidad de colores generan una buena cantidad de combinaciones suficientes para un programa simulador de este tipo.

Ejemplos del cambio de colores se notan en las figuras: 9, 1, 2, 3 y 4

2.5.6 Proceso de evolución

La función para el cálculo de la siguiente generación se encuentra en los módulos *CellularAutomaton* y *CUDACellularAutomaton* de las sección *Código Fuente*.

El proceso del cálculo de una nueva generación consiste en el cálculo del estado de todas las células que conforman al espacio a través de la contabilización del número de células vecinas en la vecindad de Moore de la célula ancla(Célula central en la ventana llamada vecindad de Moore con tamaño 3x3).

Esta función es sin duda el algoritmo más relevante en toda la simulación, y que requiere de un par de recursos para ser calculada:

- Espacio toroidal: Este recurso describe el comportamiento del espacio de evolución durante el cálculo de una nueva generación, en el que al igual que un toroide(volúmen geométrico con forma de dona) se enlazan los extremos de las filas y columnas del espacio. Este recurso se utiliza para asegurar la existencia de la vecindad de Moore en cada una de las células, siendo especificamente beneficiadas todas las células que existen en las filas y columnas extremos del espacio.
  Existen algunas alternativas para lograr este comportamiento, más la solución utilizada en este trabajo fue la adición de un *padding*(Filas y columnas extras en los extremos de un *array*) en el que se replican los valores de los extremos contrarios, resultando en una amplicación del *array* del espacio de evolución en 2 filas y 2 columnas en las primeras 2 dimensiones.

- Regla de evolución: Definida en una tupla cuadruple de valores enteros, las reglas de evolución dictan las condiciones en las que el estado de una célula cambia(Permanece viva, Muere o Nace).
  El primer par de valores funciona como un rango del número de células vecinas vivas para que la célula ancla permanezca viva, mientras el último par es el rango de células vecinas vivas bajo las que una célula ancla Muerta cambie su estado a Viva(Nacimiento).

Este proceso se lleva a cabo en una función que itera sobre todos los elementos del espacio de evolución()con excepción de los elementos en la última fila y última columna), esto se debe a que la conformación de las vecindades de Moore se realizan tomando como célula ancla la célula ubicada 1 columna y 1 fila siguiente del elemento que realmente localiza la tupla de las variables iteradoras *x,y*. A partir de que se conforma una ventana de 3x3, se algoritmo básico explicado al principio de la sección, se contabiliza el número de vecinos, se aplica la regla de evolución en función del estado actual de la célula ancla y el número de vecinos vivos, y finalmente el estado de la nueva célula se guarda en la misma ubicación bidimensional pero en el segundo elemento del *array* tridimensional, conservando el estado anterior del espacio de evolución en el primer elemento de la tercer dimensión en el *array*. Cuando se obtiene el nuevo valor de la célula, se actualiza inmediatamente el estado y color de la célula gráfica en la interfaz, conservando la congruencia entre los espacios lógicos y gráficos. Finalmente cuando se han calculado todos los nuevos estados de las células en la nueva generación, se trasladan los nuevos valores de la nueva generación al primer elemento de la 3° dimensión del *array*(pasan a formar parte de los valores de la generación actual), y se retornan a 0 los valores del segundo elemento en la 3° dimensión.

Este proceso es ligeramente diferente cuando se utiliza una mejora del rendimiento con GPU, pues el array que se copia desde el dispositivo al *host* no es como tal el espacio de evolución, sino un *array* que indica en que posiciones las células gráficas deben cambiar su estado actual unicamente, pues como ya se ha mencionado anteriormente los *array* de espacio de evolución actualizados permanecen en la memoria de la GPU.

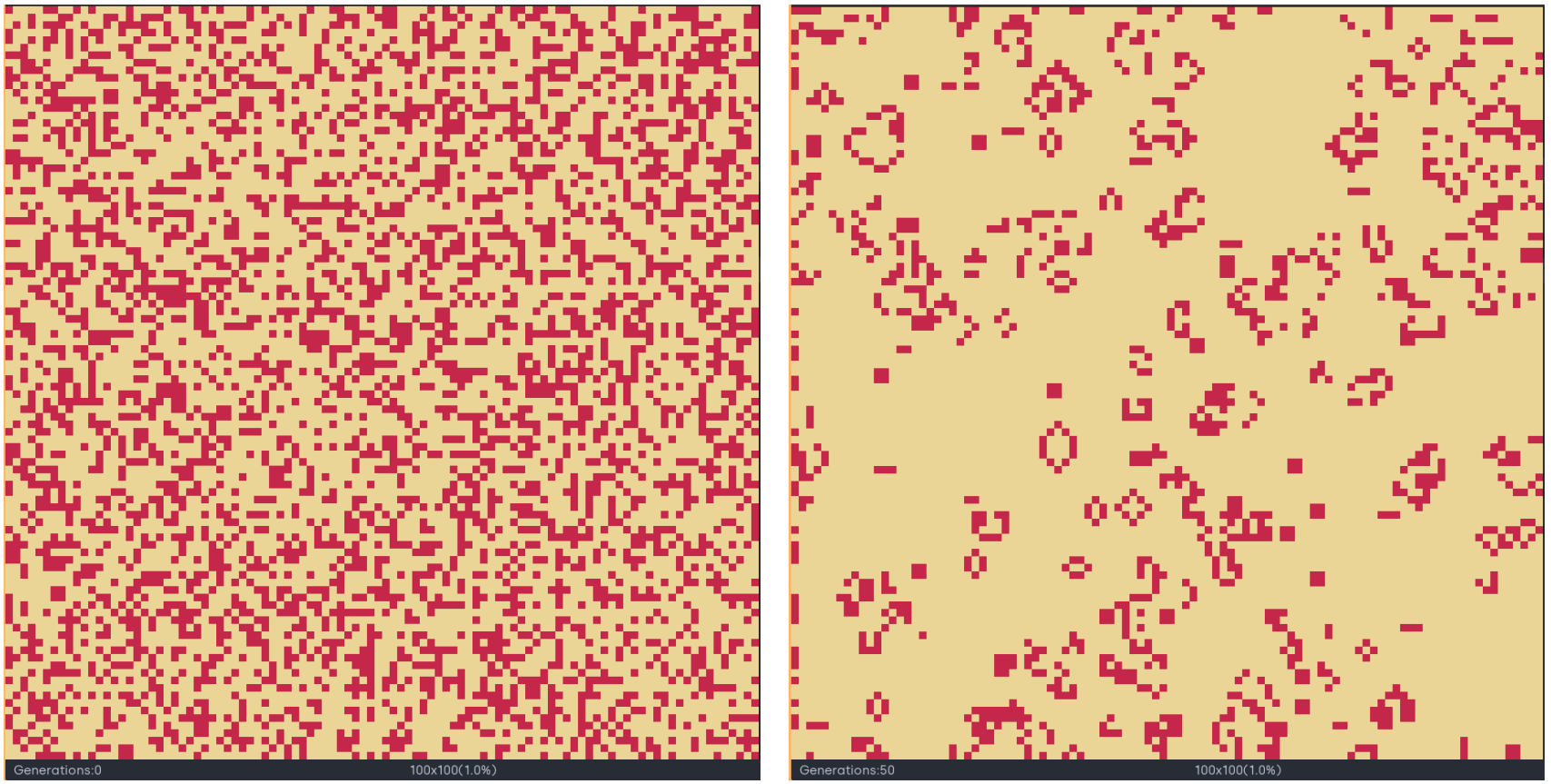

Table 1: Ejemplo de la evolución de un espacio de tamaño 100x100 después de 50 Generaciones al partir de una configuración inicial con 45% de probabilidad para el estado 0
Colores: Amarillo → Muertas, Rojo → Vivas

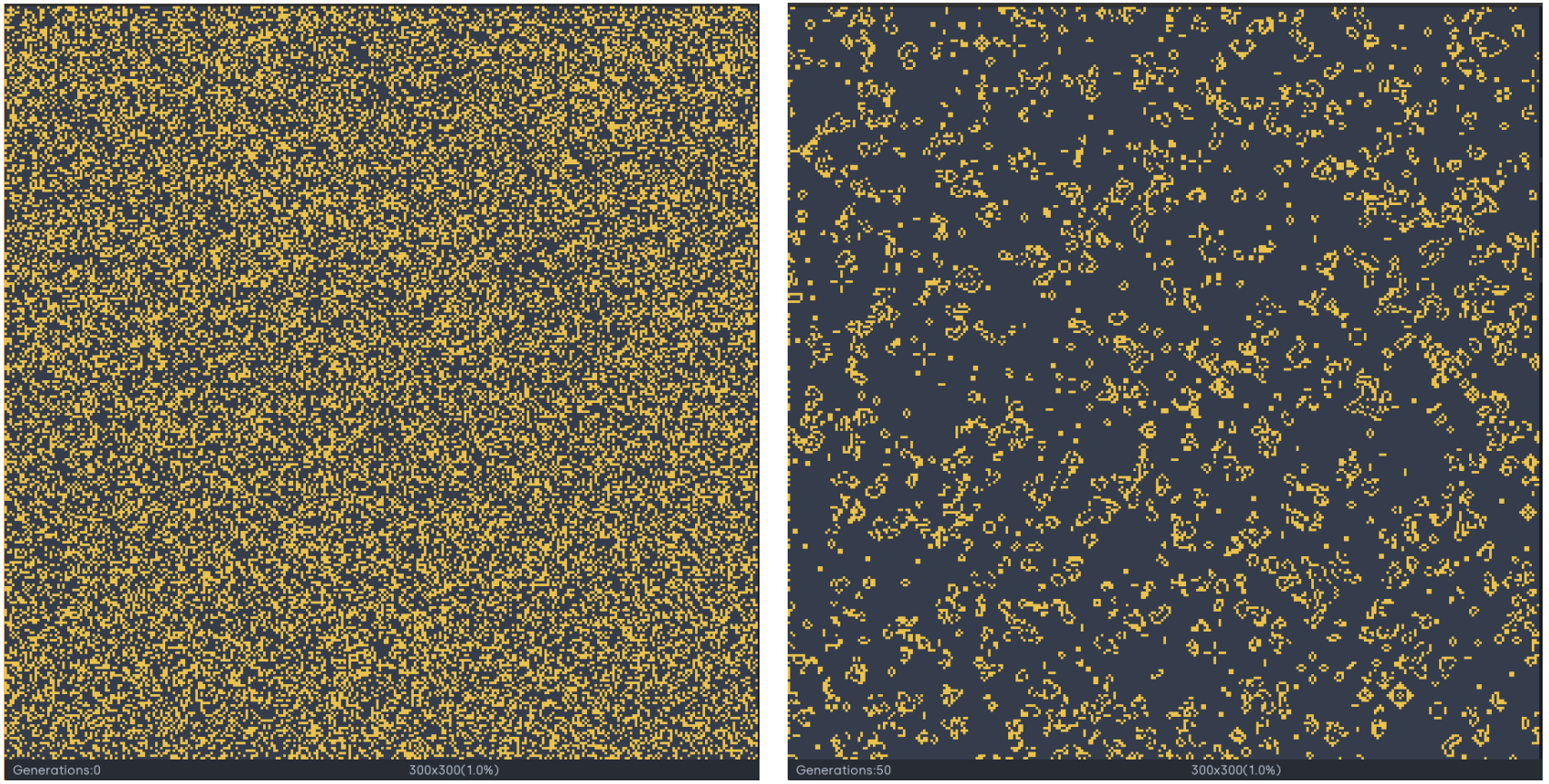

Table 2: Ejemplo de la evolución de un espacio de tamaño 300x300 después de 50 Generaciones al partir de una configuración inicial con 45% de probabilidad para el estado 0
Colores: Negro → Muertas, Amarillo → Vivas

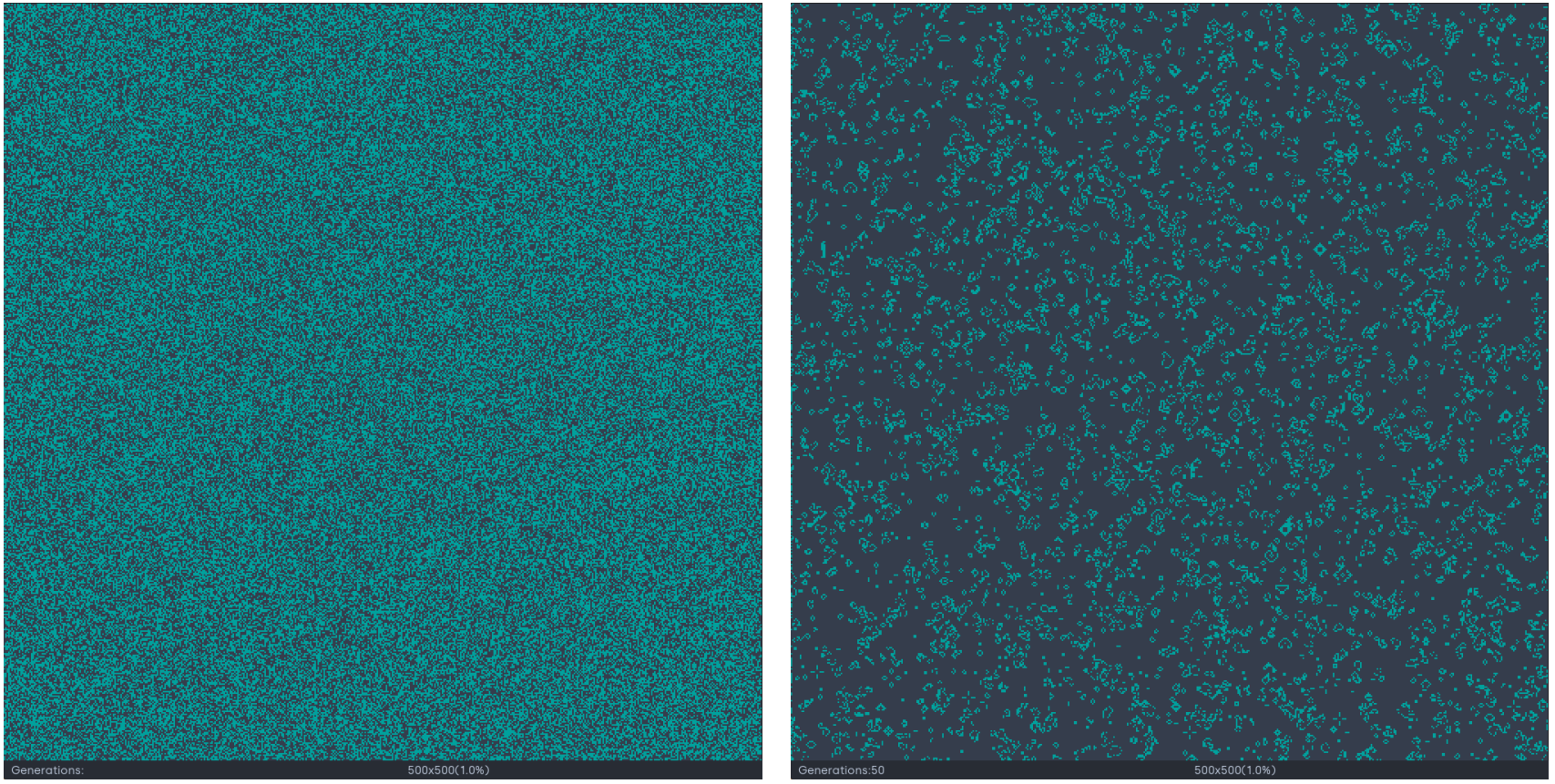

Table 3: Ejemplo de la evolución de un espacio de tamaño 500x500 después de 50 Generaciones al partir de una configuración inicial con 45% de probabilidad para el estado 0
Colores: Negro → Muertas, Azul → Vivas

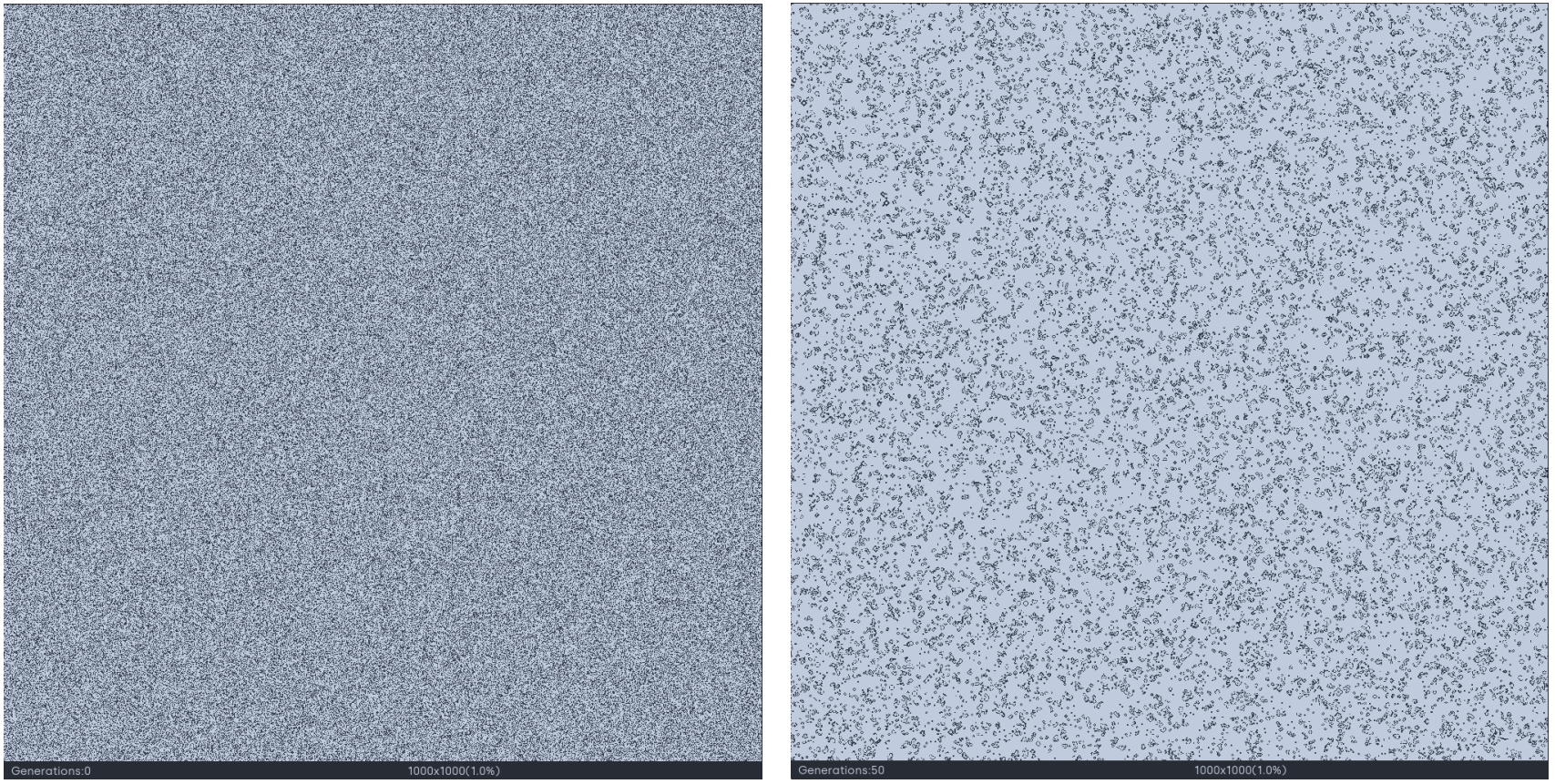

Table 4: Ejemplo de la evolución de un espacio de tamaño 1000x1000 después de 50 Generaciones al partir de una configuración inicial con 45% de probabilidad para el estado 0
Colores: Negro → Muertas, Blanco → Vivas

2.6 Graficación

Parte de un análisis estadístico de los espacios de evolución, se tiene la graficación de parámetros que ayuden a describir de alguna u otra forma al espacio de evolución, y en más específico la generación actual.

Se habilita en la simulación la graficación de 3 parámetros mediante el uso de 3 botones en su respectiva sección(Figura 4)

2.6.1 Densidad

Primer botón del arreglo de 3. Este butón graficará cuando presionado, el número de células vivas(densidad) de todas las generaciones evolucionadas hasta el momento.

Su implementación es bastante sencilla, requiriendo únicamente de una lista a la que se le agregan el número de células nuevas después de calcular una nueva evolución.

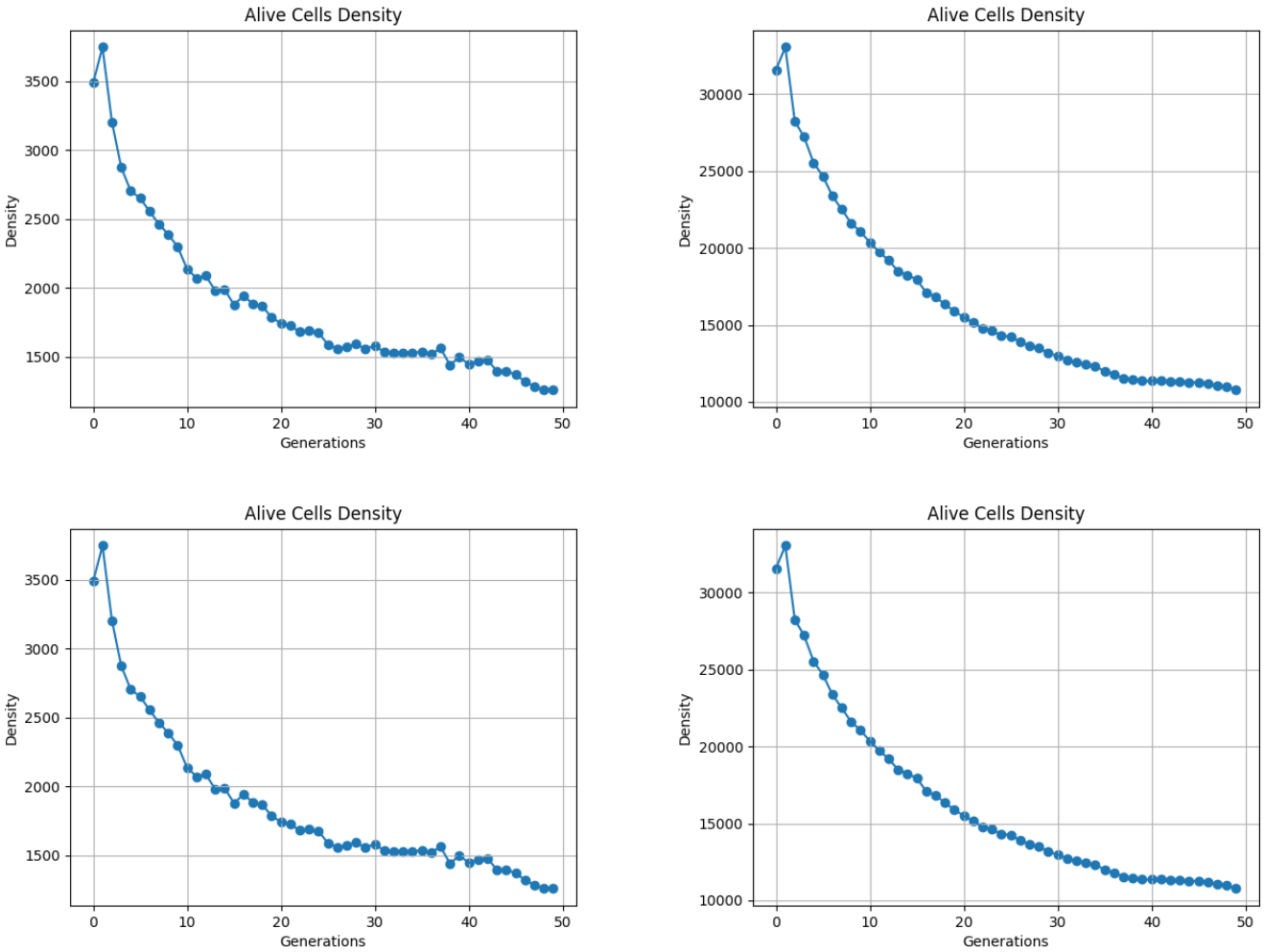

Table 5: Gráficas de densidad después de 50 generaciones para diferentes tamaños de espacios de evolución iniciadas con configuraciones aleatorias con 45% de probabilidad para el estado 1
Tamaños de los espacios de evolución, de izquierda a derecha y arriba hacia abajo: a) 100x100, b) 300x300, c) 500x500, d) 1000x1000

2.6.2 Logaritmo Densidad

Mismo funcionamiento e incluso parámetro que la gráfica de densidad, con la única particularidad de graficarse el logaritmo de esta: $log_{10}(Densidad)$

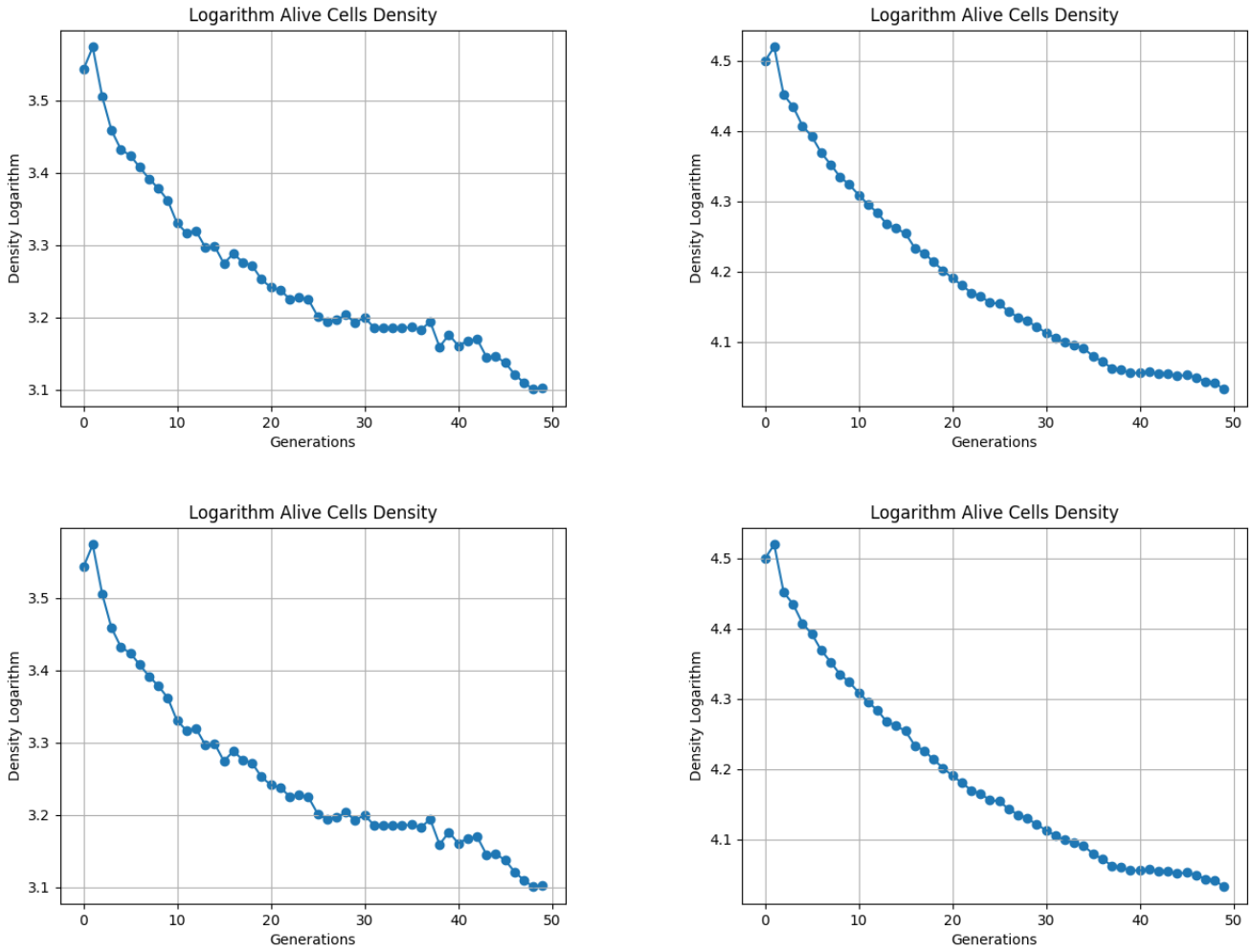

Table 6: Gráficas del logaritmo de la densidad después de 50 generaciones para diferentes tamaños de espacios de evolución iniciadas con configuraciones aleatorias con 45% de probabilidad para el estado 1
Tamaños de los espacios de evolución, de izquierda a derecha y arriba hacia abajo: a) 100x100, b) 300x300, c) 500x500, d) 1000x1000

2.6.3 Entropia

La entropía de Shannon en el ámbito de la información se trata de una cantidad que indica le incertidumbre de una fuente de información, también pudiendo interpretarse como la cantidad de información promedio que contienen los símbolos utilizados. Los símbolos con menor probabilidad de aparición serán los que aporten mayor información a la medida, de forma que llegan aumentar o disminuir el valor de esta.

Cuando se tenga una distribución homogenea de los símbolos existentes(cada símbolo diferente tendrá la misma probabilidad), entonces la entropía toma el valor de 1, mientras más símbolos diferentes en distribución heterogea existan en un conjunto, más alejado el valor de la entropía se encontrará del 1.

Partiendo del predicado "La entropía mide la cantidad de certidumbre en la incertidumbre", resulta natural asocial la certidumbre de algo que pase con la probabilidad P, entonces la incertidumbre debe ser el inverso de la Probabilidad $\frac{1}{P}$. La multiplicación de los dos términos satisface el predicado, sin embargo si se implementara de esta forma, obtendríamos que cuando un símbolo tiene probabilidad 1 la medida de incertidumbre sería igualmente 1, pero lo correcto sería que este fuera 0 pues estamos completamente seguros que cualquier símbolo escogido siempre será el mismo, por esta razón se implementa el logaritmo:

$$H = \sum p(x) log\left(\frac{1}{p(x)}\right)$$

Aplicando propiedades de logaritmos:

$$H = \sum p(x)(log(1) - log(p(x))) = \sum (p(x)(0) - p(x)log(p(x)))$$

Resultando finalmente en la ecuación planteada por Claude Shannon en 1948:

$$H = -\sum p(x)log(p(x))$$

A partir de esta ecuación lo que se quiere implementar para su graficación en la simulación, es calcular el desorden o deferencias, en las vecindades existentes en un espacio de evolución después de calcular su nueva generación. Para esto primero se requiere contabilizar la frecuencia de aparación de cada tipo de vecindad en el espacio, la solución utilizada para esta contabilización fue la de asignar un número único a cada tipo de vecindad, y la forma más sencilla es tomando a los elementos de la vecindad como un número binario, donde naturalmente la posición de cada elemento está ponderado y le corresponde una potencia de 2 para su conversión a decimal. Al convertir esta matriz con sus valores a un valor decimal, este valor sirve como id que identifica al tipo de vecindad permitiendo el incremento de su frecuencia cuando contabilizado.

Después de la contabilización de frecuencias de cada vecindad se realiza la división por el número de elementos en el espacio, obteniendo así la probabilidad de aparición de cada vecindad en esa generación para el espacio de evolución.

Finalmente tan solo resta aplicar la ecuación de la Entropía de Shannon y mediante un for calcular la entropía de esta vecindad, restándola al resultado general.

Afortunadamente para el contexto de la optimización, el cálculo de la frecuencia de las vecindades en un espacio de evolución es un algoritmo paralelizable(sucede lo mismo para el cálculo final del entropía por cada vecindad, sin embargo este proceso se realiza en CPU pues requiriría el transporte de un *array* con 512 elemento de la memoria *host* a la GPU cada vez que se calculara, resultando en tiempo de procesamiento muy iguales tan solo por esta acción), significando que es posible su cálculo con una función *Kernel* en la GPU. Esta función y su algoritmo de ejecución es explicada a detalle en la Subsección **Aceleración por GPU**, limitándose en esta sección a indicar su uso cuando la variable de configuración aceleración por GPU se encuentra activa.

Un ejemplo de los resultados obtenidos de la graficación de la Entropía para un espacio de evolución después de algunas generaciones se observa en la figura 7

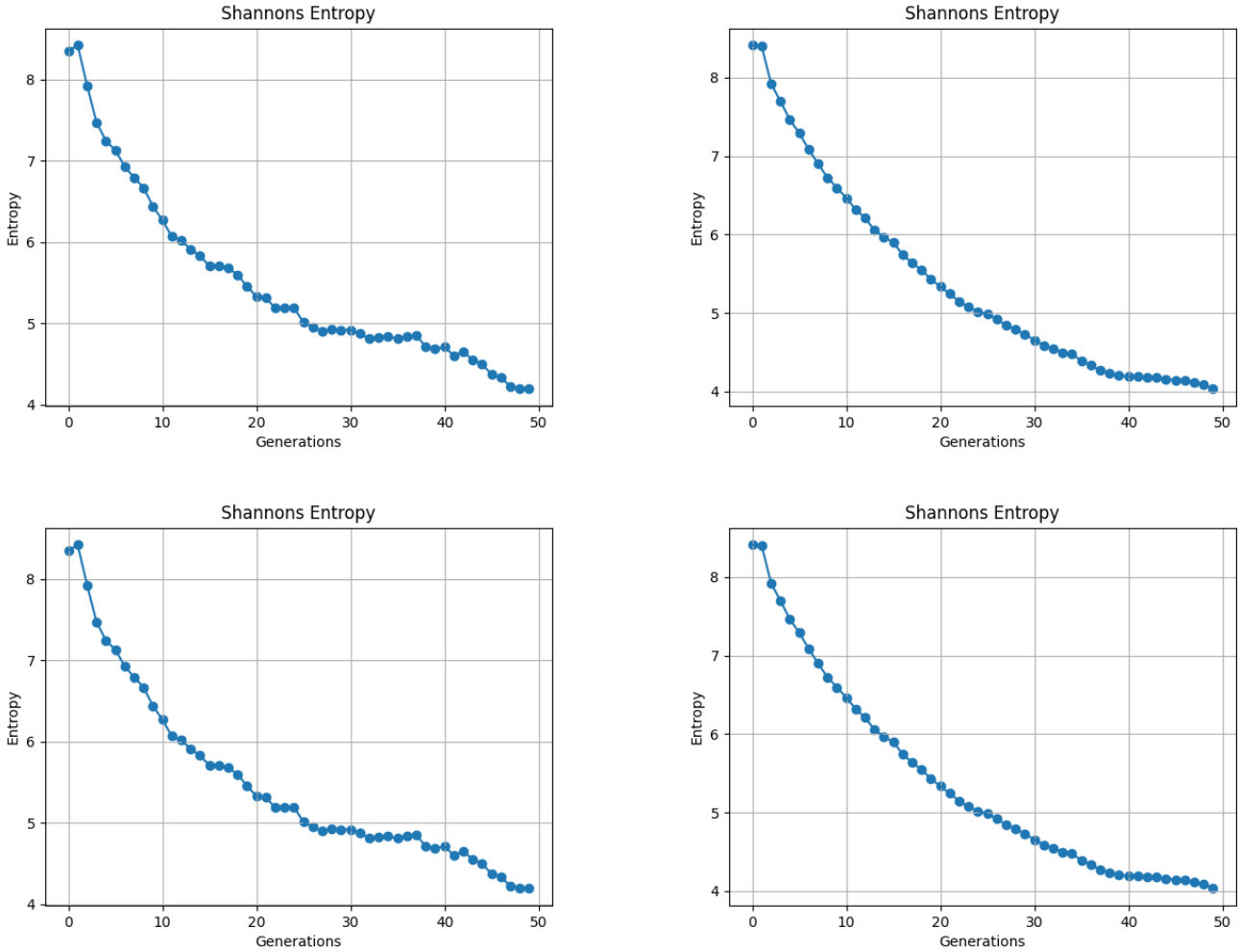

Table 7: Gráficas de la entropía de Shannon para las vecindades después de 50 generaciones para diferentes tamaños de espacios de evolución iniciadas con configuraciones aleatorias con 45% de probabilidad para el estado 1
Tamaños de los espacios de evolución, de izquierda a derecha y arriba hacia abajo: a) 100x100, b) 300x300, c) 500x500, d) 1000x1000

3 Conclusión

El objetivo del documento es el de mostrar el funcionamiento de la aplicación creada como simulador de evoluciones de espacios de autómatas en 2D. Inspirado principalmente en el concepto y funcionamiento del Juego de la Vida de Conway, se logró exitosamente implementar un formato de regla de tuplas de 4 valores, expandiendo el conjunto de posibles reglas y por lo tanto comportamientos de simulaciones.

De los requisitos funcionales necesarios planteados para la creación del simulador, se lograron cumplir con todos ellos, siendo únicamente un requisito no funcional el que ha quedado pendiente de implementación y es el añadido de una función de Zoom. Este único requisito es algo complejo de implementar tomando en cuenta las pocas herramientas disponibles en PyGame para su desarrollo, implicando que su implementación y diseño sea completamente partiendo desde 0.

Se resalta que del simulador, se trabajo en el desarrollo de los componentes gráficos aún con las pocas herramientas que ofrece la biblioteca PyGame, requiriendo en su gran parte su diseño desde 0. Todo esto se realizó tratando de cumplir el objetivo de implementar un simulador que no solo funcional, fuera intuitivo para utilizar y estéticamente agradable en su interfaz gráfica.

Importante señalar que en un principio del desarrollo el aspecto de la optimización no presentaba gran dificultad por lo reducido de las dimensiones de los espacios con los que se trataba, así como los limitados componentes gráficos, aún cuando por sus diferentes funciones desencadenadas por eventos, los componentes como botones y el *slider* requieren de un consumo de tiempo de ejecución en CPU para validación y ejecución de estos eventos. Sin embargo a medida que el simulador crecía en componentes y se iniciaban las pruebas con dimensiones mayores cumpliendo con los requisitos de espacios de 1000x1000, el desempeño se empezó a mostrar como detrimento mayor en la ejecución del simulador.

Una alternativa que resulto ser útil, al menos ahorrando ciclos de procesamiento en la ejecución de procesos pertenecientes a la lógica y otorgando resultados rápidos en tiempo de operaciones pesadas, grandes y tardadas sobre *arrays*, fue la paralelización de procesos y el uso de la GPU para la optimización del simulador. Aún con esta gran carga liberada del CPU, en dimensiones grandes el simulador aún muestra un desempeño un poco reducido pero bastante aceptable, su razón fue identificada en el refresco frecuente de todos los elementos gráficos presentes en la interfaz, más sin embargo esta oportunidad de mayor optimización no pudo ser tratada en este simulador, al menos en la última versión existente en la fecha que se escribe el documento, pero ciertamente me permitió concientizarme de esta importante modificación, obligando que desde un principio el simulador siguiente de Autómatas Celulares Elementales en 1D, sean implementadas técnicas especializadas de optimización gráfica otorgando resultados importantes con posibilidades de manejo de grandes espacios de evolución con apenas diferencia en rendimiento de espacios más pequeños.

Este simulador además de permitir comprender y explorar la búsqueda de la solución para la evolución de los autómatas celulares en 2D regidos por la regla de la vida, me brindó la oportunidad de estudiar soluciones de optimización mediante paralelización de funciones con los *Kernels* de una tarjeta gráfica, permitiéndome conseguir el aprendizaje de una herramienta sumamente útil en otras ramas de la carrera que también forman parte de mi interés profesional, como sucede con el *Machine Learning* donde la paralelización se utiliza para el entrenamiento de modelos y redes neuronales en el *Deep Learning*, o también para operaciones sobre imágenes en la rama de Análisis de Imágenes y Visión por Computadora, etc.

4 Código Fuente

El código fuente también puede ser encontrado en el repositorio de GitHub para su consulta o descarga para prueba: **https://github.com/LaloValle/Game-of-life**

El código fuente está conformado por un total de 6 módulos de Python y un archivo principal donde se cuenta con la función main que controla el flujo principal del programa y contexto gráfico para la biblioteca PyGame.

4.1 Módulos

Los módulos son archivos de código Python que agrupan clases, funciones, constantes, etc. que semánticamente se encuentran correlacionadas bajo un contexto común. Los siguientes módulos contienen en su mayoría Clases que definen componentes lógicos, nodos de un sistema, componentes gráficos; también pueden contener Constantes de uso extendido entre los módulos del programa.

4.1.1 Constant

Módulo que integra algunos recursos constantes comúnes a varios de los módulos en la simulación. Los recursos incluyen tuplas de colores en formato RGB, instancias de fuentes para PyGame, matrices de conversión, y reglas predefinidas para el proceso de evolución.

```
import pygame
from numpy import array

# Colors
DARK_BLACK = (28,29,32)
LIGHT_BLACK_1 = (40,44,52)
LIGHT_BLACK_2 = (53,61,76)
# Gray pair
WHITE = (192,203,221)
GRAY = (162,170,185)
# Blue pair
BLUE = (17,192,186)
LIGHT_BLUE = (168,234,232)
# LIGHT_BLUE = (79,189,186)
# Yellow pair
YELLOW = (234,194,76)
LIGHT_YELLOW = (234,213,150)
# Red pair
RED = (196,39,73)
LIGHT_RED = (234,159,175)
# Green pair
GREEN = (0,161,157)
LIGHT_GREEN = (79,189,186)

COLORS_LIST =
    [DARK_BLACK,LIGHT_BLACK_1,LIGHT_BLACK_2,WHITE,GRAY,BLUE,LIGHT_BLUE,YELLOW,LIGHT_YELLOW,RED,LIGHT_RED,GREEN,LIGHT_GR


# Fonts
pygame.font.init()
FONT = pygame.font.SysFont('Silka',15,False,False)
SMALL_FONT = pygame.font.SysFont('Silka',12,False,False)

# Wighted matrix for binary to decimal conversion
MATRIX_BIN_TO_DEC = array([
    [1 ,  2 ,  4],
    [8 ,  16 ,  32],
    [64 , 128 , 256],
])

# Rules
R_Life = (2, 3, 3, 3)
R_2 = (7, 7, 2, 2)
```

4.1.2 Layouts

Parte de los módulos relacionados a los gráficos, este módulo está conformado por clases que como elementos utilizados en interfaces gráficas, funcionan como herramientas que dan estructura a los elementos o que aportan cierta funcionalidad relacionada con su estructuración. Estos elementos pueden tener, o no, una representación gráfica visual.

Clase Grid Mencionada en secciones anteriores, esta clase define un componente estructural sin representación visual, pero que provee métodos útiles para la estructuración automática de elementos en un formato tipo tabla.

Clase SideBar Elemento gráfico estructural con representación visual como de un gran rectángulo que abarca lo alto de la ventana, es utilizada para agrupar visualmente elementos gráficos, y proveer una posición de referencia para colocar otros componentes. En particual esta clase es una clase *Singleton*, de forma que en el contexto de la simulación existe una instancia única, por lo que la definición y colocación de elementos como botones, etc. Se realiza directamente en la clase.

Clase BottomBar Elemento gráfico estructural con representación visual como un delgado rectángulo ubicado en la parte inferior de la ventana ocupando todo su ancho. Con comportamiento parecido a la clase SideBar, sirve como agrupador visual y como referencia para otros componentes, siendo en este caso el de los textos que indican las variables dinámicas de la simulación.

```
from turtle import width
import pygame
import numpy as np
from Constant import *
import Graphics as grph
from GraphicalComponents import *


class Grid():
    """Class that defines a grid given a position, number of columns, rows and a padding
    Helps to locate easilly a list of elements in a grid layout

    Methods
    -------
    get_element_size(self) : self.element_size
        Getter for the size of the elements in the grid

    calculate_element_size(self) :
        Method that computes the size of each element depending on the number of columns
        and padding specified when creating the grid

    locale_elements(self,list_elements) :
        Locates the elements provided in the grid from left to right, and top to bottom
    """

    def __init__(self, window, size:tuple, coord:tuple=(0,0), num_cols:int=2, num_rows:int=2, padding:int=5):
        self.window = window
        self.size = np.array(size,np.uint16)
        self.coord = coord
        self.num_cols = num_cols
        self.num_rows = num_rows
        self.padding = padding
        self.element_size = self.calculate_element_size()

    def get_element_size(self):
        return self.element_size

    def calculate_element_size(self):
        # Compute of the size of each element
        return tuple(np.around(
                    (self.size -
                        np.array([self.padding*(self.num_cols+1),self.padding*(self.num_rows+1)],np.uint16))
                        / # The original size of the grid gets substracted the number of paddins by row and
                        column
```

```
                    np.array([self.num_cols,self.num_rows],np.uint16)) # The resultant array gets divided
                        between the number of columns and rows respectively
            )

    def locate_elements(self,list_elements):
        y_pos = self.padding+self.coord[1] # Initial position with just the padding added to the original y
        # Loop in number of rows
        for row in range(self.num_rows):
            x_pos = self.padding+self.coord[0] # Initial position with just the padding added to the oiriginal x
            # Loop in number of columns
            for column in range(self.num_cols):
                index = row*self.num_cols+column
                if index < len(list_elements): list_elements[index].move((int(x_pos),int(y_pos)))
                x_pos += self.padding+self.element_size[0] # The next element must be placed in the x position
                    with added padding and width of the previous element
            y_pos += self.padding+self.element_size[1] # The next row of elements must be placed in the y
                position with added padding and width of the previous elements


class SideBar():
    """Simple rectangle used as side bar

    Methods
    -------
    draw(self,window) :
        Draws a rectagle at the right side with a given width and background color
    """

    # Singleton Class
    side_bar = None

    PLAY_BUTTON = 0
    STOP_BUTTON = 1
    RESTART_BUTTON = 2
    CLEAR_BUTTON = 3
    DRAG_SLIDER_BUTTON = 4
    DENSITY_BUTTON = 5
    DENSITY_LOGARITHM_BUTTON = 6
    ENTROPY_BUTTON = 7
    ALIVE_COLOR_INPUT = 8
    DEAD_COLOR_INPUT = 9
    SAVE_BUTTON = 10
    UPLOAD_BUTTON = 11

    def __init__(self,color_background:tuple,width:int=250,bottom_margin:int=0):
        # The only instance is the first created in the main
        SideBar.side_bar = self

        self.color_background = color_background
        self.width = width
        self.bottom_margin = bottom_margin
        self.padding = 15

        # The position and size gets define according to the windows size
        window_size = pygame.display.get_window_size()
        self.rectangle = pygame.Rect(
            ( window_size[0]-width, 0 ),
            ( width, window_size[1]-bottom_margin )
        )

        # Graphical elements inside the bar
        self.graphical_elements = []
        self.slider = self.set_slider()
        self.graphical_elements.append(self.slider)
        # Sections
```

```
        self.plot_section = Section(self.rectangle.x,180,self.rectangle.width,self.padding,'PLOTTING')
        self.colors_section =
            Section(self.rectangle.x,285+self.padding,self.rectangle.width,self.padding,'COLORS')
        self.graphical_elements.append(self.plot_section)
        self.graphical_elements.append(self.colors_section)
        # Graphical sprites inside the bar
        self.graphical_sprites = pygame.sprite.Group()
        self.graphical_sprites.add(self.set_play_button())
        self.graphical_sprites.add(self.set_stop_button())
        self.graphical_sprites.add(self.set_restart_button())
        self.graphical_sprites.add(self.set_clear_button())
        self.graphical_sprites.add(self.slider.get_drag_button())
        self.graphical_sprites.add(self.set_density_button())
        self.graphical_sprites.add(self.set_logarithm_button())
        self.graphical_sprites.add(self.set_entropy_button())
        self.graphical_sprites.add(self.set_alive_color_input())
        self.graphical_sprites.add(self.set_dead_color_input())
        self.graphical_sprites.add(self.set_save_button())
        self.graphical_sprites.add(self.set_upload_button())


    def set_click_function(self,button,function):
        self.graphical_sprites.sprites()[button].click_function = function

    def set_play_button(self):
        window_size = pygame.display.get_window_size()
        # Size of the button
        radius = 20
        image_size = 25
        # The position in the center horizontally and at the top
        x = window_size[0] - self.width/2 - radius
        y = self.padding + radius + 5

        return CircularButton(x,y,radius,image_size,image_path='./images/play_w.png')

    def set_stop_button(self):
        window_size = pygame.display.get_window_size()
        # Size of the button
        radius = 20
        image_size = 15
        x = window_size[0] - self.width/2 - 2*radius - 15*2
        y = self.padding + radius + 5

        return CircularButton(x,y,radius,image_size,image_path='./images/stop_w.png')

    def set_restart_button(self):
        window_size = pygame.display.get_window_size()
        # Size of the button
        radius = 20
        image_size = 20
        x = window_size[0] - self.width/2 + 1.5*radius
        y = self.padding + radius + 5

        return CircularButton(x,y,radius,image_size,image_path='./images/restart_w.png')

    def set_clear_button(self):
        window_size = pygame.display.get_window_size()
        # Size of the button
        radius = 18
        image_size = 13
        x = window_size[0] - 2*radius - self.padding
        y = self.rectangle.height - 2*radius - 2*self.padding
        return CircularButton(x,y,radius,image_size,border=False,image_path='./images/clear_w.png')

    def set_density_button(self):
```

```
        # Size of the button
        radius = 30
        image_size = 30
        x = self.rectangle.x + self.padding
        y = 180 + 30 + self.padding

        return CircularButton(x,y,radius,image_size,border=False,image_path='./images/density_w.png')

    def set_logarithm_button(self):
        # Size of the button
        radius = 30
        image_size = 30
        x = self.rectangle.x + self.padding + 8 + 2*radius
        y = 180 + 30 + self.padding

        return CircularButton(x,y,radius,image_size,border=False,image_path='./images/logarithm_w.png')

    def set_entropy_button(self):
        # Size of the button
        radius = 30
        image_size = 30
        x = self.rectangle.x + self.padding + 16 + 4*radius
        y = 180 + 30 + self.padding
        return CircularButton(x,y,radius,image_size,border=False,image_path='./images/entropy_w.png')

    def set_slider(self):
        width = self.rectangle.width - 2*self.padding - 10
        x = self.rectangle.x + self.padding + 5
        y = 100

        return Slider(x,y,width)

    def set_alive_color_input(self):
        x = self.rectangle.x + self.padding + 5
        y = self.colors_section.header_rect.y + 30 + self.padding
        background_color = grph.GameGraphics.cells_colors[1]
        input =
            Input(x,y,45,label='Alive',width=FONT.size('Alive')[0]+50,allow_focus=False,background_color=background_colo
        input.set_click_function(self.change_alive_cell_color)
        return input

    def set_dead_color_input(self):
        x = self.rectangle.x + 2*self.padding + 5 + 95
        y = self.colors_section.header_rect.y + 30 + self.padding
        background_color = grph.GameGraphics.cells_colors[0]
        input =
            Input(x,y,45,label='Dead',width=FONT.size('Dead')[0]+50,allow_focus=False,background_color=background_color)
        input.set_click_function(self.change_dead_cell_color)
        return input

    def set_save_button(self):
        window_size = pygame.display.get_window_size()
        # Size of the button
        radius = 18
        image_size = 13
        x = window_size[0] - 4*radius - 8 - self.padding
        y = self.rectangle.height - 2*radius - 2*self.padding

        return CircularButton(x,y,radius,image_size,border=False,image_path='./images/save_w.png')

    def set_upload_button(self):
        window_size = pygame.display.get_window_size()
        # Size of the button
        radius = 18
        image_size = 13
```

```
        x = window_size[0] - 6*radius - 16 - self.padding
        y = self.rectangle.height - 2*radius - 2*self.padding

        return CircularButton(x,y,radius,image_size,border=False,image_path='./images/upload_w.png')

    # Functions when colors of cells clicked
    def change_dead_cell_color(self):
        input = self.graphical_sprites.sprites()[SideBar.DEAD_COLOR_INPUT]
        index_color_actual = COLORS_LIST.index(input.background_color)
        new_color = COLORS_LIST[index_color_actual+1 if index_color_actual < len(COLORS_LIST)-1 else 0]
        grph.GameGraphics.cells_colors[0] = new_color
        input.background_color = new_color
        grph.GameGraphics.game_graphics.group_cells.update()

    def change_alive_cell_color(self):
        input = self.graphical_sprites.sprites()[SideBar.ALIVE_COLOR_INPUT]
        index_color_actual = COLORS_LIST.index(input.background_color)
        new_color = COLORS_LIST[index_color_actual+1 if index_color_actual < len(COLORS_LIST)-1 else 0]
        grph.GameGraphics.cells_colors[1] = new_color
        input.background_color = new_color
        grph.GameGraphics.game_graphics.group_cells.update()

    def move_drag_button(self,x):
        self.slider.move_drag_button(x)

    def stop_button_pressed(self,button):
        self.graphical_sprites.sprites()[button].pressed_once = False

    def draw(self,window):
        pygame.draw.rect(window,self.color_background,self.rectangle)
        # All the sprite elements get draw
        for element in self.graphical_elements:
            element.draw(window)
        self.graphical_sprites.draw(window)
        # Inputs cells colors
        self.graphical_sprites.sprites()[SideBar.DEAD_COLOR_INPUT].draw(window)
        self.graphical_sprites.sprites()[SideBar.ALIVE_COLOR_INPUT].draw(window)

    def get_graphical_sprites(self):
        return self.graphical_sprites


class BottomBar():
    """Simple rectangle used as bottom bar

    Methods
    -------
    draw(self,window) :
        Draws a rectagle at the bottom with a given height and background color
    """

    # Singleton Class
    bottom_bar = None

    def __init__(self,color_background:tuple,height:int=35):
        BottomBar.bottom_bar = self

        self.color_background = color_background
        self.height = height

        # The position and size gets define according to the windows size
        window_size = pygame.display.get_window_size()
        self.rectangle = pygame.Rect(
            ( 0, window_size[1]-height ),
            ( window_size[0], height )
```

```
        )

        # Creation of texts
        self.texts = []
        self.generations_text = Text((10,3+self.rectangle.y),'Generations:')
        self.alive_cells_text = Text((self.rectangle.width-30-FONT.size('Alive Cells: 0')[0],
            3+self.rectangle.y),'Alive Cells: 0')
        self.space_dimension_zoom_text = Text((self.rectangle.width/2-FONT.size('100x100(1.0)')[0],
            3+self.rectangle.y),'100x100(1.0)')
        self.texts.append(self.generations_text)
        self.texts.append(self.alive_cells_text)
        self.texts.append(self.space_dimension_zoom_text)

    def draw(self,window):
        pygame.draw.rect(window,self.color_background,self.rectangle)

        for txt in self.texts:
            txt.print(window)

    def update_generations(self,generations):
        self.generations_text.update('Generations:'+str(generations))

    def update_alive_cells(self,alive_cells):
        self.alive_cells_text.update('Alive Cells: '+str(alive_cells))

    def update_space_dimension_zoom(self,space_dimension,zoom):
        self.space_dimension_zoom_text.update('{}x{}({}%)'.format(space_dimension,space_dimension,zoom))
```

4.1.3 GraphicalComponents

Módulo relacionado a los gráficos, integrando de hecho clases que definen componentes gráficos con representación visual.

Control Definida como una clase modelo o abstracta, define el formato o esqueleto básico de muchos de los componentes *Sprite* en el mismo módulo.

MousePointer Única clase de componente gráfico sin representación visual en este módulo. Su principal objetivo es permitir identificar el colapso de otros componentes *Sprite* dentro de la interfaz con el cursor.

Consiste primordialmente en un *Sprite* rectángulo de una tamaño generalmente pequeño como de 1px por 1px, que se mueve a la misma posición que sigue el cursor.

CircularButton Clase que representa un componente gráfico con visualización, tiene generalmente un forma redondeada de botón. Esta trata de imitar la estética y funcionalidad de un botón con funciones al desencadenarse un evento(hover, click, stop_click, etc.)

Text Clase de componente gráfico que incluye texto en la interfaz gráfica. Esta clase provee métodos para el manejo sencillo del texto

Slider Clase de componentes gráfico compuesto con representación visual. El principal objetivo de esta clase es modelar la estética y funcionamiento normal de un *Slider* encontrado en cualquier interfaz grafica, proveyendo métodos para el manejo de eventos, etc.

Input Clase de componentes gráfico compuesto con representación visual. El principal objetivo de esta clase es modelar la estética y funcionamiento normal de un *Input* encontrado en cualquier interfaz grafica y más comunmente en formularios, proveyendo métodos para el manejo de eventos, etc.

Section Clase de componente gráfico con representación visual. Este componente funge únicamente como separadador visual de secciones, y aunque podría utilizarse como referencia para la colocación de elementos, su objetivo principal es visual y nadamás.

```
import pygame
import threading
from Constant import *

class Control(pygame.sprite.Sprite):

    def __init__(self):
        super().__init__()
        self.pressed_once = False

    def update(self):
        pass

    def hover(self):
        pass

    def exit(self):
        pass

    def click(self):
        pass

    def stop_click(self):
        pass


class MousePointer(pygame.sprite.Sprite):
    """Simple class that provides an invisible rect for the pointer
    Used for easily identify collisions with the mouse pointer

    Methods
    -------
    update(self) :
        Updates the position of the rect in the coordenates of the mouse
    """

    def __init__(self,size:tuple):
        super().__init__()

        self.image = pygame.Surface(size)
        self.rect = self.image.get_rect()

    def update(self):
        self.rect.x, self.rect.y = pygame.mouse.get_pos()


class CircularButton(Control):

    def
         __init__(self,x,y,radius:int,image_size:int,image_path:str='',border:bool=True,background:tuple=DARK_BLACK,hover
        super().__init__()
        self.id = id
        self.actual_background = background
        self.button_background = background
        self.button_hover_background = hover_background
        self.button_click_background = click_background

        self.border = border
        self.radius = radius
        self.image_size = image_size
        self.click_function = None

        # Controlls that only 1 time the button gets pressed
        self.pressed_once = False
```

```
        self.ignore_multiple_click = False

        # The main surface gets created
        self.image = pygame.Surface((radius*2,radius*2))
        self.image.fill(DARK_BLACK)
        self.rect = self.image.get_rect(x=x,y=y,width=radius*2,height=radius*2)

        # Icon of button
        self.icon = None
        if image_path:
            self.icon = pygame.image.load(image_path)
            self.icon_rect = self.icon.get_rect(x=x,y=y,width=image_size,height=image_size)
            self.icon = pygame.transform.scale(self.icon,(image_size,image_size))

        # Circle of the outline
        if border: pygame.draw.circle(self.image,WHITE,(self.radius,self.radius), self.radius,width=1)

    def update(self):
        # Background and icon
        pygame.draw.circle(self.image,self.actual_background,(self.radius,self.radius), self.radius-1 if
            self.border else self.radius)
        if self.icon != None:
            self.image.blit(self.icon,(self.radius-self.image_size/2,self.radius-self.image_size/2))

    def hover(self):
        self.actual_background = self.button_hover_background

    def exit(self):
        self.actual_background = self.button_background

    def set_ignore_multiple_click(self):
        self.ignore_multiple_click = True

    def click(self):
        if not self.pressed_once:
            self.actual_background = self.button_click_background
            action_thread = threading.Thread(target=self.click_function())
            action_thread.start()
            if not self.ignore_multiple_click: self.pressed_once = True

class Text():

    def __init__(self,position,text:str='Some Text',small_font=False):
        self.text = text
        self.position = position
        self.small_font = small_font
        if small_font: self.render = SMALL_FONT.render(self.text, True, WHITE)
        else: self.render = FONT.render(self.text, True, WHITE)

    def update(self,new_text):
        self.text = new_text
        if self.small_font: self.render = SMALL_FONT.render(self.text, True, WHITE)
        else: self.render = FONT.render(self.text, True, WHITE)

    def print(self,window):
        window.blit(self.render,self.position)


class Slider():

    def __init__(self,x:int,y:int,width:int,min:int=0,max:int=1):
        super().__init__()
        self.slider_background = DARK_BLACK
        self.corner_radius = 5
        self.width = width
```

```
        # The main surface gets created
        self.slider_surface = pygame.Surface((width,self.corner_radius*4))
        self.slider_surface.fill(DARK_BLACK)
        self.rect = self.slider_surface.get_rect(x=x,y=y,width=width,height=self.corner_radius*4)

        # Values of the slider between the limits
        self.value = 0.5

        # Coordenates values
        self.y_center = self.corner_radius*2
        self.reference_rectangle_width = self.width/2 - self.corner_radius

        # Drag circular button
        self.drag_button_group = pygame.sprite.Group()
        self.drag_button = CircularButton(self.rect.x+(self.rect.width/2)-self.corner_radius*2, self.rect.y,
            self.corner_radius*2, 0, background=WHITE, hover_background=WHITE, click_background=GRAY,
            border=False, id='Drag Button')
        self.drag_button.set_ignore_multiple_click() # Allow to drag the button while keep pressing the click
        self.drag_button_group.add(self.drag_button)

        # Base geometric shapes
        pygame.draw.circle(self.slider_surface,WHITE,( self.corner_radius,      self.y_center),
            self.corner_radius)
        pygame.draw.circle(self.slider_surface,WHITE,( self.width-self.corner_radius, self.y_center),
            self.corner_radius,width=1)
        pygame.draw.rect(self.slider_surface,WHITE,[ self.corner_radius,
            self.y_center-self.corner_radius, self.rect.width-self.corner_radius*2, self.corner_radius*2
            ],width=1)
        # Circle to hide the intersection between the right circle corner and the rect
        pygame.draw.rect(self.slider_surface,DARK_BLACK,[self.width-self.corner_radius*2,
            self.corner_radius*1.5-1, self.corner_radius*1.5, (self.corner_radius*2)-2])
        # Texts
        self.value_text = Text((self.rect.x+(self.width/2)-(FONT.size('0.500')[0]/2),
            self.rect.y+(self.corner_radius*4)+10),"0.500")
        self.texts = [
            Text((self.rect.x+5, self.rect.y+(self.corner_radius*4)+8.5),"0's",small_font=True),
            Text((self.rect.x+self.width-self.corner_radius-15,
                self.rect.y+(self.corner_radius*4)+8.5),"1's",small_font=True),
            self.value_text
        ]

        # Defines the limits of the drag button
        self.min_limit = self.rect.x
        self.max_limit = self.rect.x+self.rect.width - self.corner_radius*2


    def draw(self,window):
        for text in self.texts:
            text.print(window)

        window.blit(self.slider_surface,self.rect)
        pygame.draw.rect(window,WHITE,[self.rect.x +
            self.corner_radius,self.rect.y+self.corner_radius,self.reference_rectangle_width,self.corner_radius*2])

    def update_value(self):
        if self.reference_rectangle_width > 0.0:
            self.value = self.reference_rectangle_width/(self.width-self.corner_radius*2)
        else: self.value = 0.0
        self.value_text.update('{:.3f}'.format(self.value))

    def get_drag_button(self):
        return self.drag_button

    def move_drag_button(self,x):
```

```
        x -= self.corner_radius*2
        if (x <= self.max_limit) and (x >= self.min_limit):
            self.reference_rectangle_width = x - self.rect.x
            self.drag_button.rect.x = x
            self.update_value()

class Input(Control):

    def
        __init__(self,x,y,height,width:int=200,padding:int=5,label:str='Label',value:str='',allow_focus=False,backgroun
        super().__init__()
        # The main surface gets created
        self.image = pygame.Surface((width,height))
        self.image.fill(DARK_BLACK)
        self.rect = self.image.get_rect(x=x,y=y,width=width,height=height)

        self.on_focus = False
        self.allow_focus = allow_focus

        self.clicked_function = lambda:print('clicked')
        self.new_character_function = None
        self.already_clicked = False

        # Label
        self.label_text = label
        self.label = Text((0,(height-FONT.size(label)[1])/2),label)
        self.label.print(self.image)
        # Background rectangle
        self.background_color = background_color
        self.background_rect_rect =
            pygame.Rect(FONT.size(label)[0]+2*padding,padding,width-FONT.size(label)[0]-2*padding,height-2*padding)
        # Input text
        self.value = value
        self.value_text = Text((self.rect.x+self.background_rect_rect.x+padding,
            self.rect.y+padding),self.value)
    #
    # Control methods
    #
    def draw(self,window):
        pygame.draw.rect(self.image,self.background_color,self.background_rect_rect)
        pygame.draw.rect(self.image,BLUE if self.on_focus else LIGHT_BLACK_1 ,self.background_rect_rect,1)
        self.value_text.print(window)

    def click(self):
        if not self.already_clicked:
            self.clicked_function()
            self.already_clicked = True

    def stop_click(self):
        self.already_clicked = False

    def has_on_focus(self):
        return self.allow_focus

    #
    # Auxiliar functions
    #
    def set_on_focus(self):
        self.on_focus = True

    #
    # External configuration methods
    #
    def new_character(self,character):
        if character == -1:
```

```
            if len(self.value) : self.value = self.value[:-1]
        else: self.value += character
        self.value_text.update(self.value)
        # Executes a function if specified when a new character gets processed into the value text
        if self.new_character_function != None: self.new_character_function()

    def set_click_function(self,function):
        self.clicked_function = function

class Section():

    def __init__(self,x,y,width,padding,title):
        self.header_surface = pygame.Surface((width,30))
        self.header_surface.fill(LIGHT_BLACK_1)
        self.header_rect = self.header_surface.get_rect(x=x,y=y,width=width,height=30)

        # The title of the section
        self.title_text = Text((padding,9),title,small_font=True)
        self.title_text.print(self.header_surface)

    def draw(self,window):
        window.blit(self.header_surface,self.header_rect)
```

4.1.4 Graphics

Módulo que integra en su definición la clase **GameGraphics**, la cuál bien podría tratarse como un nodo del sistema, encargándose en específico de la coordinación gráfica entre elementos gráficos, eventos y comunicación con el nodo lógico principal de la simulación. Su comportamiento definido por los métodos y atributos que maneja podría definirla como una clase *Front-End* del programa de simulación.

Así mismo conjunta una segunda clase llamada **Cell**, siendo la definición de una célula gráphica. Esta clase hereda de la clase de PyGame *Sprite*, permitiendole verificar colisiones con otros *Sprites*, sin embargo no solo es una definición de esta clase gráphica, pues se añaden algunos métodos más que le permiten modificar su comportamiento y valores, exponiendo sus métodos como principal interfaz para la interacción con instancias de esta clase.

```
import pygame
import numpy as np
from Constant import *
from Layouts import *


class GameGraphics():

    game_graphics = None
    cells_colors = [WHITE,LIGHT_BLACK_2]

    # Singleton Class
    def get_game_graphics(number_columns:int=0 ,grid:Grid=None, side_bar:SideBar=None,
        bottom_bar:BottomBar=None):
        if GameGraphics.game_graphics == None: GameGraphics.game_graphics =
            GameGraphics(number_columns=number_columns, grid=grid, side_bar=side_bar, bottom_bar=bottom_bar)
        return GameGraphics.game_graphics

    def __init__(self, number_columns, grid:Grid, side_bar:SideBar=None, bottom_bar:BottomBar=None):
        # Interface drawable sprite elements
        self.drawable_elements = []
        self.collidable_elements = pygame.sprite.Group()

        # Grid for the cells
        self.grid = grid

        # Array of cells
        self.cells = []
        self.group_cells = pygame.sprite.Group()
```

```
        self.space_side_number_elements = number_columns
        self.create_cells(self.grid.get_element_size())

        # Mouse button states
        self.click_pressed = False
        self.changed_cells = []

        # Logical Game of Life instance
        self.cellular_automaton = None

        # Collapsable pointer
        self.pointer = MousePointer((1,1))

        # Graphical elements
        self.side_bar = side_bar
        self.bottom_bar = bottom_bar
        self.add_graphical_element( self.side_bar )
        self.add_graphical_element( self.bottom_bar )
        self.collidable_elements.add(self.side_bar.get_graphical_sprites())
        # Set of the elements that must be executed their respective exit function
        self.next_exit_elements = set()
        self.next_stop_pressed = set()

    def reset(self):
        # Mouse button states
        self.click_pressed = False
        self.changed_cells = []
        # Set of the elements that must be executed their respective exit function
        self.next_exit_elements = set()
        self.next_stop_pressed = set()
        # kills all the cells
        for cell in self.cells:
            cell.alive = False
            cell.color_cell()


    # PyGame loop related methods
    def process_events(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                return True
            if event.type == pygame.MOUSEBUTTONDOWN:
                if event.button == 1: # Left click
                    self.click_pressed = True
            elif event.type == pygame.MOUSEBUTTONUP:
                if event.button == 1: # Left click
                    self.click_pressed = False
                    # The 'status_changed_click' variable in the cells returns to False
                    while self.changed_cells: self.changed_cells.pop().status_changed_click = False
            if event.type == pygame.MOUSEWHEEL:
                print(event)
            # Looks for key press
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_n:
                    self.cellular_automaton.compute_next_generation()
                    # Statistical analysis
                    self.cellular_automaton.density()
                    self.cellular_automaton.density_logarithm()
                    self.cellular_automaton.shannon_entropy()
                    print('<--- Next Generation --->')
        return False

    def run_logic(self):
        self.pointer.update()
        self.collidable_elements.update()
```

```
        # Verifies if a collapsible element of the UI collided
        collaided_elements = pygame.sprite.spritecollide(self.pointer,self.collidable_elements,False)


        # If the click is pressed then the cells collapsed change status
        if self.click_pressed:
            self.change_cells_status()
        # if stop clicking
        else:
            # If there's elements that where pressed
            if self.next_stop_pressed:
                for element in self.next_stop_pressed:
                    element.pressed_once = False
                    element.stop_click()
                self.next_stop_pressed.clear()
            # If there's elements that collided while not clicked
            elif self.collidable_elements and self.next_exit_elements:
                remove_element = None
                for element in self.next_exit_elements:
                    # Identifies if the element no longer collides with the pointer
                    if element not in collaided_elements:
                        element.exit()
                        remove_element = element
                # The elements get removed from the set
                if remove_element != None:
                    self.next_exit_elements.remove(remove_element)

        # Verifies the state of those elements collided
        for element in collaided_elements:
            if self.click_pressed:
                element.click()
                self.next_stop_pressed.add(element)
            elif not self.click_pressed:
                element.hover(); self.next_exit_elements.add(element)

    def display_frame(self,window):

        window.fill(LIGHT_BLACK_2)

        self.group_cells.draw(window)
        for graphical_element in self.drawable_elements:
            graphical_element.draw(window)

        pygame.display.flip()

    # Auxiliar methods
    def add_graphical_element(self,element):
        self.drawable_elements.append(element)

    def create_cells(self,cell_size):
        # The number of cells gets created
        for row in range(self.space_side_number_elements):
            for column in range(self.space_side_number_elements):
                cell_aux = Cell(cell_size,(column+1,row+1)) # The columns and rows gets added 1 to compensate
                    the padding of the logic array
                self.cells.append(cell_aux)
                self.group_cells.add(cell_aux)
        # The cells get located in the grid
        self.grid.locate_elements(self.cells)

    def change_cells_status(self):
        for cell in pygame.sprite.spritecollide(self.pointer, self.group_cells, False):
            if cell.change_status():
                self.changed_cells.append(cell)
```

```
            # The kill/born of the cell get's also changed in the logic
            self.cellular_automaton.change_cell(cell.index,cell.alive)

    # External communications methods
    def set_cellular_automaton(self,cellular_automaton):
        self.cellular_automaton = cellular_automaton

    def get_cells(self):
        return self.cells

    def update_cell_status(self,status,cell_position,invert:bool=False):
        cell_aux = self.cells[ cell_position[0]*self.space_side_number_elements + cell_position[1]]
        # When invert active changes the state of the cell to the contrary of the current cell state
        if invert: cell_aux.set_status(not cell_aux.alive)
        # Specifies an specific status for the cell
        else: self.cells[ cell_position[0]*self.space_side_number_elements + cell_position[1]].set_status(True
            if status == 1 else False)




class Cell(pygame.sprite.Sprite):

    def __init__(self, size:tuple, index:tuple, alive:bool=False):
        super().__init__()

        self.status_changed_click = False
        self.alive = alive
        # Index of the cell in the general grid
        self.index = index
        # The cell and the rect inside gets defined
        self.image = pygame.Surface(size)
        self.image.fill(GameGraphics.cells_colors[int(self.alive)])
        # The rect of the cell is asigned so in
        # further methods can be used to move it
        self.rect = self.image.get_rect()

    def update(self):
        # Updates the background color of the cell
        self.image.fill(GameGraphics.cells_colors[int(self.alive)])

    def resize(self,size:tuple):
        self.rect.width, self.rect.height = size

    def move(self,coord:tuple):
        self.rect.x = coord[0]
        self.rect.y = coord[1]

    def color_cell(self):
        self.image.fill(GameGraphics.cells_colors[int(self.alive)])

    def set_status(self,status):
        self.alive = status
        self.color_cell()

    def change_status(self):
        if not self.status_changed_click:
            self.alive = not self.alive
            self.color_cell()
            self.status_changed_click = True
            return True
        return False
```

4.1.5 CellularAutomaton

Módulo se integra únicamente de la clase **CellularAutomaton**, clase que por definición de las responsabilidades, métodos y atributos que maneja, puede ser considerado uno de los Nodos principales de la simulación. Esta clase se encarga del manejo de la lógica detrás de la simulación, definiéndose en esta métodos encargados de la configuración, evolución y comunicación externa para los espacios de evolución. Evidentemente y al igual que sucede con la clase **GameGraphics**, tiene una relación directa de comunicación con el nodo principal gráfico para mantener la consistencia de los espacios de células lógicos y gráficos.

Los métodos de esta clase principalmente se enfocan en la creación, configuración, reinicio y evolución del espacio de autómatas, integrando a su vez métodos auxiliares que se exponen como interfaces de comunización con el nodo gráfico de la simulación. Se cuenta también con métodos encargados de la graficación de las diferentes mediadas estadísticas de los espacios, y el guardado y alzado de espacios desde archivos CSV.

```
"""
================================================================================
author: Luis Eduardo Valle Martinez
date: March 14th of 2022
subject: Complex Systems
================================================================================
"""

import math
import time
from math import log10
from numpy import *
from random import random
from Constant import MATRIX_BIN_TO_DEC,R_Life,R_2
from matplotlib import pyplot as plt

from CUDACellularAutomaton import *
from Graphics import GameGraphics
from Layouts import BottomBar

class CellularAutomaton():
    """Class that represents the parameters and elements of the Conway's Game of Life
    The behavior of each cell in the space is addressed by the 4 main essential rules:
        1. Any live cell with fewer than two live neighbours dies(underpopulation)
        2. Any live cell with two or three live neighbours lives on to the next generation
        3. Any live cell with more than three live neighbours dies(overpopulation)
        4. Any dead cell with exactly three live neighbours borns as a live cell(reproduction)

    Attributes
    ----------
    dimensions : tuple
        Tuple of the dimensions of the grid

    grid : array
        2D array of the grid space

    states : dict
        The 2 possible states [0,1] used as keys, for a tuple value that represents
        first: The top limit of the probability of generation for the random initial
        configuration, and second: The color used to depict the cell state

    generation : int
        Number of the actual generation. By default starts with the value 0

    alive_cells
    newly_born_cells
    newly_deceased_cells

    Methods
    -------
    random_initial_config():
        Creates a random initial configuration taking in count the probability of each
```

```
    state for its assingment
"""

def __init__(self,size,use_gpu,rule:tuple=R_Life):

    self.dimensions = (size,size,2)
    self.number_cells = size*size
    self.actual_rule = rule
    # Grid of 3 dimensions
    # Mimicking the rows and columns respectively, the first 2 dimensions
    # are the same, meanwhile the 3rd has 2 elements being the first used
    # for the actual cell grid states, and the 2nd element to calculate
    # the new grid for the next generation
    self.space = zeros(self.dimensions,ubyte)
    # The padding at the first and last rows and columns is needed for the
    # window of 3x3, used to count the number of neigbours of the anchor element
    self.add_padding()
    # States
    self.zeros_density = 0.5
    # Dynamic variables through the process
    self.generations = 0
    self.alive_cells = 0
    # Graphics connection
    self.game_graphics = None

    # Record of statistical analysis
    self.density_record = []
    self.density_logarithm_record = []
    self.shannon_entropy_record = []

    # If enhancement of GPU
    self.ca_gpu = None
    self.gpu_enhancement = use_gpu
    if use_gpu:
        print('<--- Enhancement by GPU active --->')
        self.ca_gpu = CUDACellularAutomaton(self.space,self.dimensions)

#
# Configuration functions
#
def reset(self):
    # Common actions for GPU and CPU implementations
    self.update_alive_cells(self.alive_cells*-1)
    self.update_generations(self.generations*-1)
    self.density_record.clear()
    self.density_logarithm_record.clear()
    self.shannon_entropy_record.clear()
    self.space = zeros(self.dimensions,byte)
    self.add_padding()
    # GPU actions
    if self.gpu_enhancement:
        self.ca_gpu.clear()

def set_game_graphics(self, game_graphics:GameGraphics):
    self.game_graphics = game_graphics

def add_padding(self):
    rows_padding = zeros((self.dimensions[0],1,2),ubyte) # 1 row with the same number of columns and shape,
        except for the 2nd dimension that's only 1
    columns_padding = zeros((1,self.dimensions[1]+2,2),ubyte) # 1 column at the first dimension with same
        number of rows+2 and same elements in 3rd dimension
    self.space = concatenate((rows_padding,self.space,rows_padding),axis=1)
    self.space = concatenate((columns_padding,self.space,columns_padding),axis=0)
    self.toroid_padding()
```

```
def random_initial_config(self, graphic_cells) -> array:
    # Loop that runs all the space elements and generates randomly an alive or dead cell
    for y in range(1,self.dimensions[1]+1):
        for x in range(1,self.dimensions[0]+1):
            if  self.zeros_density > random.random(): # Probability of an alive cell is the complement of
                the state 0
                self.update_alive_cells(1)
                self.space[y,x,0] = 1
                graphic_cells[(y-1)*self.dimensions[0]+(x-1)].set_status(True)

    # GPU actions
    if self.gpu_enhancement:
        self.ca_gpu.initial_configuration(self.space[:,:,0],self.alive_cells,self.actual_rule)

#
# Update of dynamic variables and the texts showed in the interface
#
def update_alive_cells(self,added_cells:int):
    self.alive_cells += added_cells
    BottomBar.bottom_bar.update_alive_cells(self.alive_cells)

def update_generations(self,added_generations:int):
    self.generations += added_generations
    BottomBar.bottom_bar.update_generations(self.generations)

def update_zeros_density(self,density):
    self.zeros_density = density

#
# Functions while running the cellular automaton
#
def define_cell_status(self,alive_neighbours, anchor_cell):
    # Substracts the anchor cell value because the method
    # recives the alive_neighbours count before taking in
    # count the value of the anchor cell
    alive_neighbours -= anchor_cell

    if not anchor_cell: # Dead cell
        if alive_neighbours >= self.actual_rule[2] and alive_neighbours <= self.actual_rule[3]:
            return 1 # A cell borns
        else:
            return 0 # The cell keeps as dead

    # Alive cell
    if anchor_cell:
        if alive_neighbours < self.actual_rule[0]: # Undepopulation
            return -1 # The cell dies in the next generation
        if alive_neighbours > self.actual_rule[1]: # Overpopulation
            return -1 # The cell dies in the next generation
        else:
            return 0 # The cell lives

def toroid_padding(self):
    self.space[0] = self.space[-2] # The first padding row is the last row with valid cells
    self.space[-1] = self.space[1] # The last padding row is the first row with valid cells
    self.space[:,0] = self.space[:,-2] # The first padding column is the last column with valid cells
    self.space[:,-1] = self.space[:,1] # The last padding column is the first column with valid cells

def compute_next_generation(self):
    time_start = time.time()
    # CPU process
    if not self.gpu_enhancement:
        self.toroid_padding()
        for y in range(self.dimensions[1]):
            for x in range(self.dimensions[0]):
```

```
                window = copy(self.space[y:y+3,x:x+3,0])
                alive_neighbours = window.sum()
                new_status_cell = self.define_cell_status(alive_neighbours, copy(self.space[y+1,x+1,0]))
                # The alive cells counter gets updated
                self.update_alive_cells(new_status_cell)
                # The new status of the cell gets saved in the second column of the 3rd dimension
                self.space[y+1,x+1,1] = new_status_cell
                # Only when the new_status cell changes then
                if new_status_cell != 0: self.game_graphics.update_cell_status(new_status_cell,(y,x))
        # Once the new states have been calculated, the new changes are applied
        self.space[:,:,0] = self.space.sum(axis=2)
        self.space[:,:,1] -= self.space[:,:,1] # The second columns returns to zeros
    # GPU process
    else:
        changed_cells = self.ca_gpu.next_generation()
        # Look for the cell changed so that it can be show in the interface
        for y in range(self.dimensions[1]):
            for x in range(1,self.dimensions[0]):
                if changed_cells[y+1,x+1] == 1: self.game_graphics.update_cell_status(True,(y,x),invert=True)
        # Updates the alive cells
        self.update_alive_cells(self.ca_gpu.get_alive_cells() - self.alive_cells)

    # Increments the generations
    self.update_generations(1)

    print('>> Time for compute_next_generation({}):
         {:.3f}s'.format(self.generations,time.time()-time_start))


#
# Communication with external methods
#
def get_generations(self):
    return self.generations

def change_cell(self,index,alive:bool):
    # The alive cells counter and graphical elements gets updated
    self.update_alive_cells(1 if alive else -1)

    # CPU actions
    if not self.gpu_enhancement:
        self.space[index[1],index[0],0] = int(alive)
        if (1 in index) or (self.dimensions[0] in index): self.toroid_padding()

    # GPU actions
    else:
        self.ca_gpu.change_cell(index)

#
# Statistical analysis
#
def density(self):
    self.density_record.append(int(self.alive_cells))

def density_logarithm(self):
    self.density_logarithm_record.append(log10(self.alive_cells))

def shannon_entropy(self):
    entropy = 0
    neighbourhood_frecuency = [ 0 for i in range(512) ]

    # CPU actions
    if not self.gpu_enhancement:
        for y in range(self.dimensions[1]):
            for x in range(self.dimensions[0]):
```

```
                window = copy(self.space[y:y+3,x:x+3,0])
                neighbourhood_number = (window*MATRIX_BIN_TO_DEC).sum()
                neighbourhood_frecuency[neighbourhood_number] += 1
        # GPU actions
        else:
            neighbourhood_frecuency = self.ca_gpu.shannon_entropy()

        probability = 0.0
        for frecuency in neighbourhood_frecuency:
            if frecuency: # Different of 0
                probability = frecuency/self.number_cells
                entropy -= probability*math.log2(probability)

        self.shannon_entropy_record.append(entropy)

    def plot_density(self):
        if not self.density_record:
            print('<!!! No density of cells has been recorded !!!>')
            return False

        fig = plt.figure()
        ax = plt.axes()

        ax.set( xlabel = 'Generations',
                ylabel = 'Density',
                title = 'Alive Cells Density'
            )
        ax.grid()

        data = list(self.density_record)
        data = array(list(enumerate(list(data))))
        ax.scatter(data[:,0],data[:,1])
        ax.plot(data[:,0],data[:,1])

        plt.show()

    def plot_density_logarithm(self):
        if not self.density_logarithm_record:
            print('<!!! No logarithm density of cells has been recorded !!!>')
            return False

        fig = plt.figure()
        ax = plt.axes()

        ax.set( xlabel = 'Generations',
                ylabel = 'Density Logarithm',
                title = 'Logarithm Alive Cells Density'
            )
        ax.grid()

        data = list(self.density_logarithm_record)
        data = array(list(enumerate(list(data))))
        ax.scatter(data[:,0],data[:,1])
        ax.plot(data[:,0],data[:,1])

        plt.show()

    def plot_shannons_entropy(self):
        if not self.shannon_entropy_record:
            print('<!!! No shannons entropy has been recorded !!!>')
            return False

        fig = plt.figure()
        ax = plt.axes()
```

```
        ax.set( xlabel = 'Generations',
                ylabel = 'Entropy',
                title = 'Shannons Entropy'
            )
        ax.grid()

        data = list(self.shannon_entropy_record)
        data = array(list(enumerate(list(data))))
        ax.scatter(data[:,0],data[:,1])
        ax.plot(data[:,0],data[:,1])

        plt.show()

    #
    # Save and upload of generations
    #
    def save_evolution_space(self):
        filename = './saves/CA_generation_{}.csv'.format(self.generations)
        if self.gpu_enhancement: self.space[:,:,0] = self.ca_gpu.get_space()
        savetxt(
            filename,
            copy(self.space[1:-1,1:-1,0]),
            delimiter = ', ',
            fmt = '% s'
            )
        print('<--- File successfully saved as \"'+filename+'\" --->')

    def upload_evolution_space(self):
        aux_array = genfromtxt("./saves/upload.csv",delimiter=', '); aux_shape = aux_array.shape
        if (self.dimensions[0]-aux_shape[0] < 0) or (self.dimensions[1]-aux_shape[1] < 0):
            print('!!! The actual evolution space is smaller than the intended upload file !!!')
        # The upload array gets loaded in the programm
        else:
            # First the space gets cleaned
            self.reset()
            self.game_graphics.reset()
            if self.gpu_enhancement: self.ca_gpu.clear()
            initial_column = 1 + int((self.dimensions[1]-aux_shape[1])/2); initial_row = 1 +
                int((self.dimensions[0]-aux_shape[0])/2)
            # The new array gets saved in the saving_space and the evolution_space arrays
            self.space[initial_row:initial_row+aux_shape[0],initial_column:initial_column+aux_shape[1],0] =
                aux_array
            self.toroid_padding()
            # The graphical cells gets updated with the value of the new array
            for y in range(initial_row,initial_row+aux_shape[0]):
                for x in range(initial_column,initial_column+aux_shape[1]):
                    self.game_graphics.update_cell_status(bool(self.space[y,x,0]),[(y-1),(x-1)])
            # Dynamic variables
            self.alive_cells = int(aux_array.sum())
            self.generations = 0
            if self.gpu_enhancement:
                self.ca_gpu.initial_configuration(self.space[:,:,0],self.alive_cells,self.actual_rule)
            print('<--- New configuration successfully uploaded --->')
```

4.1.6 CUDACellularAutomaton

Módulo final de la simulación. Creado a partir del rendimiento limitado que se conseguia cuando se trabajaban con espacios de evolución de gran tamaño, buscando liberar carga de procesamiento del CPU y acelerar el tiempo de ejecución de algoritmos pesados en procesamiento y tiempo de ejecución.

El módulo esta conformado por las funciones *Kernel* y una clase que controla su ejecución.

Kernel and CUDA Functions Se ha desarrollado más extensamente cada una de las funciones definidas y su funcionamiento específico en la sección **Aceleración por GPU**, más no se encuentra de más recordar que estas funciones son

implementaciones paralelizadas de procesos generalmente ejecutadas de forma secuencial en la CPU cuando se manejan estructuras de varios elementos como los *array*. Utilizando la tecnología CUDA de Nvidia, se permiten crear un número de bloques con un número de hilos por cada bloque, siendo habitual la definición de tantos hilos como elementos en el *array*, implicando que el código de la función sea tomando en cuenta la operación con 1 elemento de ese array pero por la paralelización la GPU estará trabajando el mismo código en cada uno de los elementos del *array* en un teórico ciclo de reloj para el procesamiento secuencial del CPU.

CUDACellularAutomaton Clase principalmente usada como interfaz de comunicación entre el nodo lógico(**CellularAutomat** y las funciones aceleradas por hardware. También permite la coordinación y ejecución de funciones paralelizadas específicas según el estado y acciones solicitadas por el usuario en la simulación.

```
import time
from numpy import *
from numba import cuda
from Constant import MATRIX_BIN_TO_DEC

#
#   Kernels and CUDA functions
#
@cuda.jit(device=True)
def cuda_define_cell_status(alive_neighbours, anchor_cell, rule):
    if not anchor_cell: # Dead cell
        if alive_neighbours >= rule[2] and alive_neighbours <= rule[3]:
            return 1 # A cell borns
        else:
            return 0 # The cell keeps as dead

    # Alive cell
    if anchor_cell:
        if alive_neighbours < rule[0]: # Undepopulation
            return 0 # The cell dies in the next generation
        if alive_neighbours > rule[1]: # Overpopulation
            return 0 # The cell dies in the next generation
        else:
            return 1 # The cell lives

@cuda.jit(device=True)
def cuda_count_neighbours(window):
    count = 0
    for y in range(3):
        for x in range(3):
            count += window[y,x]
    # If count greater than 0, gets the anchor cell value substracted
    if count > 0: count -= window[1,1]
    return count

@cuda.jit
def cuda_next_generation(space,out_space,alive_cells,changes_space,rule):
    x = cuda.threadIdx.x
    y = cuda.blockIdx.x
    # If its not the last 2 rows/columns
    if (x < space.shape[0]-2) and (y < space.shape[0]-2):
        # Clears whichever the past result was in the changes_space array
        changes_space[y+1,x+1] = 0
        # Count of neighbours
        alive_neighbours = cuda_count_neighbours(space[y:y+3,x:x+3])
        # Assigns the new value of the cell
        anchor_cell = space[y+1,x+1]; new_cell_value =
            cuda_define_cell_status(alive_neighbours,anchor_cell,rule)
        out_space[y+1,x+1] = new_cell_value
        # When the status of the cell changed
        if new_cell_value != anchor_cell:
            # Puts 1 if the status of the cell changed
            changes_space[y+1,x+1] = 1
```

```
            # Updates the alive cells
            cuda.atomic.add(alive_cells, 0, 1 if new_cell_value == 1 else -1)

@cuda.jit
def cuda_update_results(space,out_space):
    x = cuda.threadIdx.x
    y = cuda.blockIdx.x
    # Evaluating if the coordinates correspond to those at the padding
    # and change it to copy the values of the borders to achieve the
    # toroid array
    x_index = -2 if x == 0 else (1 if x == (space.shape[0]-1) else x)
    y_index = -2 if y == 0 else (1 if y == (space.shape[0]-1) else y)
    space[y,x] = out_space[y_index,x_index]

@cuda.jit
def cuda_shannons_probability(space,neighbourhood_frecuency,conversion_matrix):
    x = cuda.threadIdx.x
    y = cuda.blockIdx.x
    # If its not the last 2 rows/columns
    if (x < space.shape[0]-2) and (y < space.shape[0]-2):
        neighbourhood_number = 0
        # Loops through the windows of the neighbourhood and adds to the
        # neighbhourhood_number variable the powers of 2 that corresponds
        # to a live cell. This allows to convert the neighbourhood into
        # a decimal number to be identified
        for y_nn in range(y,y+3):
            for x_nn in range(x,x+3):
                if space[y_nn,x_nn]: neighbourhood_number += conversion_matrix[y_nn-y,x_nn-x]
        # Increments the neighbourhood frecuency
        cuda.atomic.add(neighbourhood_frecuency ,neighbourhood_number, 1)

@cuda.jit
def cuda_change_cell(position,space,alive_cells):
    space[position[1],position[0]] = int(not space[position[1],position[0]])
    # Updates the alive cells
    cuda.atomic.add(alive_cells, 0, 1 if space[position[1],position[0]] else -1)

@cuda.jit
def cuda_clear_space(space,out_space):
    x = cuda.threadIdx.x
    y = cuda.blockIdx.x
    space[y,x] = 0
    out_space[y,x] = 0

class CUDACellularAutomaton():

    def __init__(self,space,dimensions):
        # Gets the shape of the space in only 2 dimensions
        self.dimensions = (0,0)
        self.alive_cells = array([0],int32)

        # Arrays in memory of the GPU
        self.space_rule = None
        self.space_device = None
        self.out_space_device = None
        self.alive_cells_device = None
        self.changes_space_device = None # Used to indicate which cells have changed after the generation
            function

    #
    # Class methods
    #
    def initial_configuration(self,space,alive_cells,rule):
        self.dimensions = space.shape
        # Restarts any memory reserved before
```

```
        self.space_device = None
        self.out_space_device = None
        self.alive_cells_device = None
        self.changes_space_device = None
        # Assigns the memory for the arrays
        self.space_rule = cuda.to_device(copy(rule))
        self.space_device = cuda.device_array_like(copy(space))
        self.out_space_device = cuda.to_device(copy(space))
        self.alive_cells_device = cuda.to_device(array([alive_cells],uint32))
        self.changes_space_device = cuda.device_array_like(copy(space))
        self.conversion_matrix = cuda.to_device(copy(MATRIX_BIN_TO_DEC))
        # Calls the kernel to update the new arrays and perform the toroidal padding assignments
        start_time = time.time()
        cuda_update_results[self.dimensions[0],self.dimensions[1]](self.space_device, self.out_space_device)
        end_time = time.time()
        print('<--- Initial configuration update ({:.6f}s) --->'.format(end_time-start_time))

    def clear(self):
        start_time = time.time()
        cuda_clear_space[self.dimensions[0],self.dimensions[1]](self.space_device,self.out_space_device)
        end_time = time.time()
        print('<--- Clear of space ({:.6f}s) --->'.format(end_time-start_time))
        # Easier to delete the existing array in the GPU memory with the alive cells count and assign a new one
        self.alive_cells_device = None
        self.alive_cells_device = cuda.to_device(self.alive_cells)

    def change_cell(self,index):
        index = array(index,int16)

        start_time = time.time()
        cuda_change_cell[1,1](index,self.space_device,self.alive_cells_device)
        end_time = time.time()
        print('<--- Change of value in cell[{},{}] ({:.6f}s) --->'.format(index[0], index[1],
            end_time-start_time))
        # print('GPU alive cells >> ', self.alive_cells_device.copy_to_host())

    def next_generation(self):
        # print(self.space_device.copy_to_host())
        start_time = time.time()
        cuda_next_generation[self.dimensions[0],self.dimensions[1]](self.space_device,self.out_space_device,self.alive_c
        cuda_update_results[self.dimensions[0],self.dimensions[1]](self.space_device, self.out_space_device)
        changes_space = self.changes_space_device.copy_to_host()
        end_time = time.time()
        print('<--- Next generation ({:.6f}s) --->'.format(end_time-start_time))
        return changes_space

    def shannon_entropy(self):
        neighbourhood_frecuency_space = cuda.to_device(array([0 for i in range(512)]))
        cuda_shannons_probability[self.dimensions[0],self.dimensions[1]](self.space_device,neighbourhood_frecuency_space
        return neighbourhood_frecuency_space.copy_to_host()

    #
    # Getters and setters
    #
    def get_alive_cells(self):
        return self.alive_cells_device.copy_to_host()[0]

    def get_space(self):
        return self.space_device.copy_to_host()
```

4.2 main.py

Archivo Python principal de la simulación, en este se maneja el cíclo principal de la aplicación gráfica, la instanciación de todos los elementos y nodos que manejan la simulación, variables y constantes de configuración general del programa.

```
import os
os.environ['PYGAME_HIDE_SUPPORT_PROMPT'] = "hide"
import pygame
from Graphics import *
from Constant import *
from CellularAutomaton import *
from Layouts import *
import time
import warnings
warnings.filterwarnings('ignore') # Hides the warnings

# Delay in ms
DELAY_IN_MS = 1
# Number of elements by side in the grid
# the total number of cell is GRID_SIDE_SIZE^2
GRID_SIDE_ELEMENTS = 50
GRID_PADDING = 0
GRID_SIZE = 800
# Elements of the interface
SIDE_BAR_WIDTH = 250
BOTTOM_BAR_HEIGHT = 25
WINDOW_TITLE = 'Game of life'
# Statistical analysis
DENSITY = True
DENSITY_LOGARITHM = True
SHANNON_ENTROPY = True
# Use GPU for enhanced performance
GPU_ENHANCEMENT = True



# Global status variables
paused = True
reset = False
clear = False
actual_zoom = 1.0
cellular_automaton = None
second_start = 0

def play():
    global paused,second_start
    paused = False
    # second_start = time.time()

def stop():
    global paused
    paused = True

def restart():
    global reset
    reset = True

def cleared():
    global clear
    clear = True

def slider_move():
    SideBar.side_bar.move_drag_button(pygame.mouse.get_pos()[0])
    cellular_automaton.update_zeros_density(SideBar.side_bar.slider.value)

def plot_shannons_entropy():
    cellular_automaton.plot_shannons_entropy()

def main():
```

```
global paused,reset,clear
global actual_zoom
global cellular_automaton

# Status variables
done = False

# Pygame configurations
pygame.init()
clock = pygame.time.Clock()

# Window configuration
window_display = 0
window_size = (GRID_SIZE+SIDE_BAR_WIDTH,GRID_SIZE+BOTTOM_BAR_HEIGHT)
window = pygame.display.set_mode(window_size,display=window_display)
pygame.display.set_caption(WINDOW_TITLE)

# Graphical elements
# The grid for the interface cells gets created
grid = Grid(window, (GRID_SIZE,GRID_SIZE), padding=GRID_PADDING, num_cols=GRID_SIDE_ELEMENTS,
    num_rows=GRID_SIDE_ELEMENTS)
side_bar = SideBar(DARK_BLACK,SIDE_BAR_WIDTH)
bottom_bar = BottomBar(LIGHT_BLACK_1,BOTTOM_BAR_HEIGHT)
game_graphics = GameGraphics.get_game_graphics(GRID_SIDE_ELEMENTS, grid, side_bar, bottom_bar)

# Logical part of the program
cellular_automaton = CellularAutomaton(GRID_SIDE_ELEMENTS,GPU_ENHANCEMENT)
cellular_automaton.random_initial_config(game_graphics.get_cells())
cellular_automaton.set_game_graphics(game_graphics)
game_graphics.set_cellular_automaton(cellular_automaton)

# Side bar functions to buttons
side_bar.set_click_function(SideBar.PLAY_BUTTON,play)
side_bar.set_click_function(SideBar.STOP_BUTTON,stop)
side_bar.set_click_function(SideBar.RESTART_BUTTON,restart)
side_bar.set_click_function(SideBar.CLEAR_BUTTON,cleared)
side_bar.set_click_function(SideBar.DRAG_SLIDER_BUTTON,slider_move)
side_bar.set_click_function(SideBar.DENSITY_BUTTON,cellular_automaton.plot_density)
side_bar.set_click_function(SideBar.DENSITY_LOGARITHM_BUTTON,cellular_automaton.plot_density_logarithm)
side_bar.set_click_function(SideBar.ENTROPY_BUTTON,cellular_automaton.plot_shannons_entropy)
side_bar.set_click_function(SideBar.SAVE_BUTTON,cellular_automaton.save_evolution_space)
side_bar.set_click_function(SideBar.UPLOAD_BUTTON,cellular_automaton.upload_evolution_space)
# Bottom bar space dimension and zoom
bottom_bar.update_space_dimension_zoom(GRID_SIDE_ELEMENTS,actual_zoom)

# Delay between each generation
delay_start = time.time()

# Main loop
while not done:
    done = game_graphics.process_events()

    game_graphics.run_logic()

    game_graphics.display_frame(window)

    clock.tick(60)

    if reset:
        cellular_automaton.reset()
        game_graphics.reset()
        # Random configuration again
        cellular_automaton.random_initial_config(game_graphics.get_cells())
        reset = False
```

```
        if clear:
            cellular_automaton.reset()
            game_graphics.reset()
            clear = False

        if not paused:
            if (time.time()-delay_start)*1000 >= DELAY_IN_MS:
                # Next generation
                if DENSITY: cellular_automaton.density()
                if DENSITY_LOGARITHM: cellular_automaton.density_logarithm()
                if SHANNON_ENTROPY: cellular_automaton.shannon_entropy()
                cellular_automaton.compute_next_generation()
                # Restarts delay start time
                delay_start = time.time()
        else: delay_start = time.time()

        # if time.time()-second_start >= 1: paused = True
        # Changes de window size
        # if cuenta == 60: window = pygame.display.set_mode((500,500),display=window_display)
        # if cuenta == 120: window = pygame.display.set_mode((1200,1000),display=window_display)
        # https://www.pygame.org/docs/ref/display.html

    pygame.quit()

if __name__ == "__main__": main()
```