FACULTAD DE INFORMÁTICA
UNIVERSIDAD POLITÉCNICA DE MADRID

UNIVERSIDAD POLITÉCNICA DE MADRID
FACULTAD DE INFORMÁTICA

TRABAJO FIN DE CARRERA

METASIMULADOR EXTENDIDO
DISEÑO, DESARROLLO Y MODELOS

AUTOR: ALBERTO GARCÍA MARRUPE
TUTOR: ANTONIO GIRALDO CARBAJO

Dedicado mis padres, Consuelo y Alberto. Su cariño es mi deuda y el cariño mi forma de pago.

Esta dedicatoria fue forjada en el mundo de las ideas y enviada a dos universos por distintos emisarios. Uno partió hacia lo que se abraza, el otro se dirigió al mundo de los recuerdos.

AGRADECIMIENTOS

Muy distintas personas, con distintos reconocimientos, se dan cita en esta apartado. Unas por razones intangibles, otras por cuestiones objetivas al proyecto y algunas por una mezcla de ambas.

Quisiera agradecer a mi madre, Consuelo, a mis hermanos, Elena y Raúl, a mi cuñado, Carlos, a mis dos sobrinos, Adrián y Alejandro, y al resto de mi familia, su paciencia cuando reflexionaba en voz alta, su energía cuando el barco parecía encallarse y por estar siempre ahí cuando uno necesitaba su pequeño mundo.

Agradezco a Tusi, en una cantidad que desconoce pero que espero que recuerde e intuya, el ánimo que despertó en mí para lograr ofrecerle un ramo de preciosas flores, creadas que no cultivadas.

Gracias a los amigos, los de siempre, por hacer de primeros espectadores críticos de esa idea inexplicable que se escribía en las servilletas de un bar.

Gracias a NoNick, al Bohemio, al Hoplita y a Big-Small-Clear por sus puntualizaciones, comentarios, correcciones, críticas constructivas e ideas con respecto a las bases del proyecto, la puesta en marcha del mismo, los calendarios de entrega, la maquetación y la impresión. Han sido la parte contratante de las primeras partes del análisis, desarrollo y pruebas de este proyecto.

Agradecer también a Antonio que confiara en esta locura, haber procurado un entorno de proyecto envidiablemente agradable y haberme permitido sustraerle algunas perlas para algunos tipos de ejemplo del simulador extendido.

Gracias a Saúl por toda su aportación a este proyecto, sin la cual nada de lo que se ve sería posible. Pero sobre todo gracias por quien es, aquello indefinible que le permite estar en cada uno de los reconocimientos anteriores.

ÍNDICE

1 RESUMEN

El proyecto presentado en esta memoria recibe el nombre de **Metasimulador extendido** y consiste en una aplicación informática capaz de representar múltiples situaciones, propias de los dominios de las Matemáticas y la Física, creadas por el usuario y notificadas al sistema mediante modelos que describen el problema a tratar.

Partiendo de la idea inicial propuesta, consistente en crear un simulador genérico que permitiera representar un vasto conjunto de situaciones teóricas y naturales en las que se vieran envueltos múltiples y variados elementos de carácter matemático o corpuscular, se ha ideado un sistema capaz de soportar tal representación en un entorno computacional asequible.

La magnitud del proyecto es tal que ha obligado a una colaboración conjunta por parte de los autores para poder disponer la base del sistema, común a este proyecto y al análogo existente. Una vez preparado el núcleo funcional, ambos proyectos han discurrido por dos caminos distintos de forma que, a la fecha de entrega, puede decirse que existe un metasimulador convencional, preparado para la simulación matemática y física basada en ecuaciones, y este metasimulador extendido, que hereda la mayoría de potencia y capacidad de su predecesor pero que es capaz de soportar simulaciones mas amplias, dentro del marco de la topología y la algorítmica, al basarse en un conjunto de funciones extendidas que permiten un número muy superior de simulaciones frente a su predecesor.

La parte conjunta de ambos simuladores es capaz de leer un modelo dado por el usuario en un archivo de configuración, basado en un sistema de funciones y variables de función, interpretarlo y visualizar dicho modelo, dentro de un espacio tridimensional de representación, en el que cada elemento existente en la ejecución se simboliza mediante un prisma hexagonal, en una posición con valores de coordenadas en el rango de los números naturales, de un tamaño dado y con una combinación de colores dada.

Esto permite al usuario asignar, para cada elemento, una posición en un momento concreto, o una velocidad y/o aceleración en un rango de tiempo dado, un tamaño para el elemento y aquellas propiedades que el usuario estime oportunas representar con los grados de color aplicables.

Aparte de estas propiedades perceptibles en la ejecución, el simulador puede gestionar tantas propiedades internas por elemento como el usuario desee incorporar y pueden ser consultadas en cualquier momento de la ejecución.

Las variables pueden tomar valores reales o enteros en cualquier magnitud computable que se desee. Las funciones transforman un conjunto de variables mediante el uso de operadores aritméticos y lógicos y funciones comunes como las trigonométricas, exponente, logaritmo decimal y neperiano, valor absoluto, parte entera y similares.

Así mismo, una función puede construirse como combinación de dos funciones distintas mediante un condicional ternario **if**, de forma que se ejecuta una u otra según se cumpla o no la expresión lógica del condicional.

Desde esta base se realizaron tres ampliaciones en el simulador original:

- **Clases**: el usuario puede agrupar los elementos en **Clases de Elemento**, que no son sino conjuntos matemáticos para los que se definen las variables y funciones necesarias, de forma que cualquier elemento de una clase dada posee las variables y funciones que posea su clase.

- **Interfaz Funcional**: sobre cada elemento o clase de elemento, se definen condiciones lógicas que, en caso de ser satisfechas, permiten la creación de nuevos elementos o la eliminación de elementos que cumplan la condición. Además permite definir una serie de coeficientes que son relacionados con una variable de nuestra elección para poder modificar su valor de partida durante la ejecución.

- **Interfaz Gráfico**: el simulador permite al usuario contemplar la ejecución desde distintos puntos de vista, puede pausar la simulación, ejecutar el modelo paso a paso o de forma continua, girar el modelo mientras se ejecuta, variar un número determinado de coeficientes del modelo, tomar instantáneas de algún paso de ejecución concreto o cargar un nuevo modelo.

Una vez se tuvieron implementadas las características funcionales descritas, se inició el desarrollo concreto del metasimulador extendido, que aumenta las prestaciones anteriores, con la inclusión de las siguientes características:

- **Condición de parada**: el simulador permite establecer condiciones lógicas de parada de forma que cada elemento que cumpla la condición deja de ser ejecutado pero se mantiene en pantalla con su última posición, radio y colores conocidos.

- **Cambio por objetivo**: un elemento puede incorporar dos grupos distintos de funciones y ejecutar las ecuaciones de uno u otro grupo en función de un objetivo dado. Si el elemento minimiza el valor de la función objetivo dada para este, entonces se ejecuta el segundo grupo de funciones y en caso contrario, se ejecuta el primer grupo.

- **Funciones extendidas**: los modelos representados en el simulador extendido permiten incorporar en las expresiones matemáticas de cualquier función llamadas a funciones extendidas. Creadas para este simulador, este grupo de funciones es capaz de calcular valores imposibles de calcular mediante los operadores y funciones matemáticas convencionales de forma que el simulador extendido es capaz de representar modelos inalcanzables para el simulador convencional. Estas funciones pueden ser ampliadas en cualquier momento con nuevas incorporaciones sin necesidad de modificar el resto del metasimulador extendido, de forma que su potencia sea considerable.

El proyecto cubre tanto los requisitos iniciales, basados en los objetivos deducibles de la idea de partida, como las diferentes etapas del desarrollo de software, a saber, análisis, diseño, implementación, verificación y pruebas.

Además, se plantean distintos ejemplos clasificados en tipos dentro de los cuales se han generado diferentes modelos para demostrar la potencia subyacente de la ampliación sobre el modelo original del sistema de partida generado.

2 INTRODUCCIÓN

Tras concluir los estudios se acercaba el momento de realizar el proyecto fin de carrera y para ello debía encontrar una propuesta satisfactoria, tanto en lo personal como para los requisitos exigidos en un trabajo de estas características.

En la gran mayoría de las conversaciones mantenidas con el coautor de este proyecto las propuestas giraban en torno a nuestro interés, predilección, por un tipo de idea que aunara matemáticas e informática y que fuera novedoso.

Abandonando la senda de los algoritmos computables de resolución de problemas matemáticos, para los que existe abundante literatura y trabajos ya realizados, nos encontramos valorando distintas versiones de juegos poco conocidos o proponiendo variaciones originales sobre alguno de ellos. El mayor obstáculo para la elección se encontraba precisamente en que la simplicidad, la falta de interés del juego o ambas, solían imponerse en el veredicto final y las propuestas terminaban por ser descartadas.

Podría decirse que las ideas propuestas hasta el momento no satisfacían la medida subjetiva del interés personal por realizar un proyecto complejo e innovador.

Tras varias semanas de búsqueda surgió la idea teórica para una aplicación capaz de soportar cualquier tipo de juego de dos contrincantes, intentando ganarlo mediante la explotación de los huecos, falta de información, en el sistema de reglas.

Bauticé este juego como **El Juego del Diablo** y expliqué a mi colega su funcionamiento con el siguiente ejemplo:

> *Un usuario ejecuta la aplicación para jugar un juego definido en sus reglas de comportamiento mediante un archivo que previamente se ha dado al sistema para su lectura y comprensión. Digamos que el archivo de juego cargado describe, mediante un lenguaje apropiado para la máquina, las siguientes reglas:*
>
> *Los jugadores se sientan a ambos extremos de un tablero y disponen de un número ilimitado de segmentos que deben colocar uno a continuación del otro o en perpendicular al último segmento puesto, de forma que logren llegar al extremo del tablero donde se encuentra su rival antes que este, bien llegando antes, bien impidiendo que el rival llegue antes.*

Para explicar mejor el concepto al que quería llegar, propuse que jugáramos una partida de este juego tomado como ejemplo, en la que quien les habla haría de máquina ejecutando la aplicación comentada. Unos palillos harían las veces de fichas del juego. Una vez listos invité a mi colega a efectuar el primer movimiento.

Mi rival puso entonces un palillo en el borde del lado de su mesa. Yo tomé otro palillo y lo puse cerca del suyo, en paralelo, de forma que un extremo del mismo tocaba el borde de su lado de la mesa y le dije: "… *he ganado* …".

Él me replicó que no podía ser que hubiera ganado ya que había puesto mi palillo en su borde inicial de la mesa y no en el opuesto a este, que era el mío. Le indiqué que estaba equivocado, ya que si repasaba las reglas del juego podría comprobar que no existía ninguna restricción al respecto de donde comenzar a poner los segmentos.

Esa falta de definición era justo la que el sistema explotaba para ganar. Es decir, buscaba un movimiento ganador mediante el análisis de la heurística existente y la detección de movimientos novedosos de acuerdo a unas malas, incompletas, reglas. Nótese que no se buscaba un programa que jugara eficazmente al juego descrito, sino que pudiera jugar eficazmente a este y a cualquier otro juego que pudiéramos describirle en términos comprensibles para una máquina.

Esta idea nos fascinó a ambos pero con el paso de las semanas, en las que analizamos en detalle su factibilidad, concluimos que era imposible su realización, al menos bajo los requisitos que nos habíamos impuesto a nosotros mismos. Con todo, la idea quedó en nuestras cabezas y creo que puede decirse que fue en parte la instigadora para llegar a la propuesta final.

Efectivamente, en algún momento en el que se estaban analizando las posibilidades reales de diseño de la aplicación, tres elementos fundamentales se dieron cita.

Nos encontrábamos pensando en la posibilidad de un sistema que admitiera ficheros de juego, lo que implicaría un lenguaje específico de definición de reglas y condiciones que un computador pudiera comprender. En segundo lugar examinábamos como debería ser la parte funcional que tomara las reglas y las ejecutara de forma adelantada para establecer que movimientos serían los ganadores y, por último, utilizamos algún juego de ejemplo diferente al de los segmentos, concretamente alguno que necesitara de un tablero de malla hexagonal en el que los elementos se representaran mediante fichas hexagonales.

Estar valorando un sistema paramétrico con ficheros de entrada en un lenguaje creado al efecto capaz de asumir reglas de juego sobre fichas hexagonales nos hizo dar el siguiente salto y plantearnos si estas fichas podrían simbolizar elementos de la materia, con reglas o comportamientos propios de esta, la física de partículas, y que estos pudieran ser los que en cada momento un usuario decidiera ejecutar mediante la carga de un fichero de configuración. En ese momento surgió la idea de realizar un simulador.

La propuesta era satisfactoria, ya que era suficientemente compleja e interesante como para abordarla. Tanto es así que, sin perder ni un ápice su interés, fue su complejidad la que hubo que negociar para poder llegar a un desarrollo viable.

Cuando el problema quedó centrado en la simulación de entidades representables mediante elementos hexagonales que disponían de variables y funciones que las modificaban, la propuesta se convirtió en un simulador matemático y físico.

Tras las primeras reuniones con nuestro tutor, otro requisito ajeno a la idea comentada debía ser tenido en cuenta. Cada uno de los autores debía proponer un proyecto netamente diferenciado aunque tuviera un inicio común.

Durante varias semanas estuve proponiéndome distintas posibilidades hasta que fue el interés por simular el algoritmo del Juego de Vida el obstáculo que definió uno de los límites de la propuesta inicial. En tanto la aplicación ejecutara modelos basados en ecuaciones lineales era del todo imposible dar cabida a algoritmos en los que se requiriese manejar conceptos como los caminos de fichas, grupos compactos de elementos o adyacencia entre elementos y huecos adyacentes a fichas.

El proyecto presentado tomó entonces rumbo a este límite para traspasarlo, consiguiendo que todos estos conceptos pasaran a ser perfectamente asumibles y, por tanto, simulables.

La ampliación de modelos quedó definitivamente diseñada cuando, en vez de crear soluciones para algunos tipos de modelo, como los comentados anteriormente, se ideó un grupo de funciones que permitieran dar cabida a estos y otros problemas y seguir extendiéndose prácticamente sin límite en el conjunto de modelos existentes.

3 OBJETIVOS

El objetivo global del proyecto consiste en crear una aplicación que permita realizar simulaciones de modelos computables extremadamente ricos y variados con tan solo introducir la información del modelo que se desee simular.

Los distintos objetivos parciales necesarios para satisfacer este objetivo global se dividen en dos grandes categorías perfectamente diferenciables, a saber:

3.1. Categoría Gráfica

Por una parte se pretende desarrollar un sistema gráfico capaz de visualizar modelos en un espacio tridimensional finito en el que puedan disponerse los distintos elementos del modelo, representados por prismas hexagonales que se mueven a posiciones discretas dadas por una malla hexagonal de tres dimensiones. La aplicación dispondrá de una funcionalidad que permita al usuario gestionar los archivos de modelos, variar el zoom, el punto de observación y distintos añadidos que mejoren la observación del modelo representado.

3.2. Categoría Matemática

La segunda categoría engloba toda la funcionalidad no gráfica del sistema y está orientada a soportar la parte computacional y matemática de la simulación de modelos.

Esta área se deberá desarrollar a lo largo del tiempo del proyecto en varios niveles incrementales que buscan aumentar progresivamente el conjunto de modelos representables hasta donde sea posible, dentro de los condicionamientos computacionales y de diseño del sistema.

Como podrá suponerse, representar cualquier modelo que seamos capaces de imaginar conlleva un análisis inicial relativo a la estructura y funcionamiento del universo de modelos, unos basados en entidades tangibles de la naturaleza, otros en modelos teóricos o abstractos dados por la matemática, otros como híbridos matemáticos que simulan comportamientos reales, modelos de autómata, modelos algorítmicos computables y una larga e interesante lista de cualidades que diferencian y categorizan cada tipología.

Este análisis debe buscar la funcionalidad mínima que permita al sistema ser capaz de representarlos, a todos o a una inmensa mayoría.

La metodología de diseño de abajo hacia arriba parece la más adecuada para construir una solución, por lo que se establecerán una serie de niveles de simulación, acorde a este análisis, que incrementen sucesivamente las capacidades del sistema.

Se proponen los siguientes niveles incrementales de simulación:

3.2.1. Primer nivel de simulación: Modelos de elementos simples

Este nivel debería ser capaz de representar modelos de elementos simples basados en sistemas de ecuaciones lineales de la forma:

Sea $X = \{x_1, x_2,.., x_n\}$ *el conjunto de variables de un elemento* E

Sea $F = \{f_1, f_2,.., f_m\}$, *donde* $m<n$, *el conjunto de funciones de un elemento* E

Sea $T = \{t_1, t_2,.., t_k,..\}$ *el conjunto de instantes de ejecución del Sistema*

Sea $x_i(t_j)$ *el valor de la variable i-ésima de un elemento* E *en el instante* t_j

Se cumple $x_i(t_{j+1}) = f_i(x_1(t_j), x_2(t_j),.., x_q(t_j))$, *donde* $q \leq n$, *para todo elemento* E

Los modelos que pueden ser representados en este nivel cuentan con un número finito de elementos, distintos entre si, a los que se les dota de ciertas propiedades y comportamientos, definidos mediante el uso de un conjunto de variables y un conjunto de funciones respectivamente.

Mediante este nivel, un usuario del sistema podrá simular modelos basados en geometría euclidiana, mecánica newtoniana y similares.

3.2.2. Segundo nivel de simulación: Clases de elementos

Este nivel deberá dotar al sistema con la capacidad de reunir distintos elementos simples de la representación en Clases de Elementos, de forma que el sistema pueda modelar comportamientos de entidades no elementales de la forma:

Sea $C = \{C_1, C_2, .., C_p\}$ *el conjunto de Clases de Elementos*

Sea $C_k = \{E_{1k}, E_{2k},.., E_{qk}\}$ *los elementos de la Clase k-ésima*

Sea $X_k = \{x_{1k}, x_{2k},.., x_{nk}\}$ *el conjunto de variables de la Clase* C_k

Sea $F_k = \{f_{1k}, f_{2k},.., f_{mk}\}$ *el conjunto de funciones de la Clase* C_k

Sea $T = \{t_1, t_2,.., t_p,..\}$ *el conjunto de instantes de ejecución del Sistema*

Sea $x_{ik}(t_j)$ *el valor de la variable ik-ésima de la Clase* C_k *en el instante* t_j

Entonces se cumple:

$$x_{ik}(t_{j+1}) = f_{ik}(x_{1k}(t_j), x_{2k}(t_j),.., x_{mk}(t_j)), \text{ donde } m<n, \text{ para todo } E_{kk}$$

con lo que un usuario del sistema podría mejorar la simulación de modelos del primer nivel al bastarle con describir el comportamiento de las clases intervinientes y declarar posteriormente cuantos elementos simples, y cuales, pertenecen a cada una de las clases empleadas en la representación.

Este nivel permitiría al usuario trabajar con modelos basados en campos de partículas, comportamiento de grupos homogéneos o cualquier modelo que disponga de elementos simples agrupados bajo idéntico comportamiento entre si.

3.2.3. Tercer nivel de simulación: Inserción y eliminación de elementos

El tercer nivel de simulación se centra en la creación y destrucción de elementos, simples o de clase, de la siguiente forma:

Sea $C_k = \{E_{1k}, E_{2k}.., E_{qk}\}$ *los elementos de la Clase k-ésima*

Sea $I_k = \{I_{1k}, I_{2k}.., I_{nk}\}$ *el conjunto de condiciones de inserción de la Clase* C_k

Sea $D_k = \{D_{1k}, D_{2k}.., D_{mk}\}$ *el conjunto de condiciones de borrado de la Clase* C_k

Entonces se cumple para la inserción:

$$El\ elemento\ E_{q+1k}\ es\ creado\ si\ (I_{1k}\hat{}I_{2k}\hat{}..\hat{}I_{nk}) = TRUE$$

y para la eliminación:

$$El\ elemento\ E_{kq}\ es\ eliminado\ si\ (D_{1k}\hat{}D_{2k}\hat{}..\hat{}D_{mk}) = TRUE$$

de esta forma, las simulaciones permiten altas y bajas de elementos, algo imprescindible en una buena parte de algoritmos como en el caso de los juegos de vida o las simulaciones de desarrollo de estructuras vegetales.

3.2.4. Cuarto nivel de simulación: Parada de elementos

El cuarto nivel de simulación establece condiciones de parada de elementos, simples o de clase, de la siguiente forma:

Sea $C_k = \{E_{1k}, E_{2k}.., E_{qk}\}$ *los elementos de la Clase k-ésima*

Sea $P_k = \{P_{1k}, P_{2k}.., P_{nk}\}$ *el conjunto de condiciones de parada de la Clase* C_k

Entonces se cumple:

$$El\ elemento\ E_{qk}\ es\ parado\ si\ (P_{1k}\hat{}P_{2k}\hat{}..\hat{}P_{nk}) = TRUE$$

Esto posibilita la representación de modelos de centenares o miles de elementos de forma eficiente ya que aquellos elementos que se encuentren parados seguirán siendo visualizados en la representación, pero no se consumirá tiempo en el cálculo de su comportamiento.

Todos los modelos fractales mediante IFS y juego de caos mejoran considerablemente si el sistema tan solo debe calcular la tupla mínima de elementos que se necesitan en cada iteración.

3.2.5. Quinto nivel de simulación: Funciones objetivo

El quinto nivel de simulación dota al sistema de la posibilidad de variar el comportamiento de cada elemento, simple o de clase, en función de hacer mínimo el valor de una función objetivo dada para cada elemento, de la siguiente forma:

Sea $C_k = \{E_{1k}, E_{2k},.., E_{qk}\}$ *los elementos de la Clase k-ésima*

Sea $X_k = \{x_{1k}, x_{2k},.., x_{nk}\}$ *el conjunto de variables de la Clase* C_k

Sea $F_k = \{f_{1k}, f_{2k},.., f_{mk}\}$ *el conjunto primario de funciones de la Clase* C_k

Sea $G_k = \{g_{1k}, g_{2k},.., g_{pk}\}$ *el conjunto secundario de funciones de la Clase* C_k

Sea $O_k = \{o_{1k}, o_{2k},.., o_{rk}\}$ *el conjunto de funciones objetivo de la Clase* C_k

Sea $T = \{t_1, t_2,.., t_s,..\}$ *el conjunto de instantes de ejecución del Sistema*

Sea $x_{ik}(t_j)$ *el valor de la variable ik-ésima de La Clase* C_k *en el instante* t_j

Sea $o_{ik}(t_j)$ *el valor de la función objetivo ik-ésima de La Clase* C_k *en el instante* t_j

Entonces:

Si $o_{ik}(t_{j-1}) < o_{ik}(t_j)$

$$x_{ik}(t_{j+1}) = \boldsymbol{f}_{ik}(x_{1k}(t_j), x_{2k}(t_j),.., x_{ok}(t_j)),\ \text{donde } o<n,\ \text{para todo } E_{kk}$$

Si $o_{ik}(t_{j-1}) \geq o_{ik}(t_j)$

$$x_{ik}(t_{j+1}) = \boldsymbol{g}_{ik}(x_{1k}(t_j), x_{2k}(t_j),.., x_{ok}(t_j)),\ \text{donde } o<n,\ \text{para todo } E_{kk}$$

este nivel nos permitiría ejecutar cualquier modelo con comportamiento dependiente de uno o varios objetivos dados al sistema.

3.2.6. Sexto nivel de simulación: Funciones Extendidas

El sexto y último nivel es sin lugar a dudas el menos convencional y el que confiere mayor potencia al sistema presentado.

Este nivel permite el uso de las llamadas funciones extendidas, es decir, funciones no estándar creadas para este sistema que permiten la representación de un vasto conjunto de tipos de modelos y, lo que es aún mejor, la incorporación de un número ilimitado de nuevas funciones que amplíen la potencia del simulador hasta límites insospechados.

Este nivel puede describirse de la siguiente forma:

Sea $C_k = \{E_{1k}, E_{2k}.., E_{qk}\}$ *los elementos de la Clase k-ésima*

Sea $X_k = \{x_{1k}, x_{2k}.., x_{nk}\}$ *el conjunto de variables de la Clase* C_k

Sea $F_k = \{f_{1k}, f_{2k}.., f_{mk}\}$ *el conjunto de funciones elementales de la Clase* C_k

Sea $S_k = \{s_{1k}, s_{2k}.., s_{pk}\}$ *el conjunto de funciones extendidas de la Clase* C_k

Sea $T = \{t_1, t_2.., t_q..\}$ *el conjunto de instantes de ejecución del Sistema*

Sea $x_{ik}(t_j)$ *el valor de la variable ik-ésima de La Clase* C_k *en el instante* t_j

Entonces:

$$x_{ik}(t_{j+1}) = [\boldsymbol{f}_{ik}(x_{1k}(t_j), x_{2k}(t_j),.., x_{qk}(t_j))] + [a * \boldsymbol{s}_{ik}(x_{1k}(t_j), x_{2k}(t_j),.., x_{qk}(t_j))]$$

donde $p,q < n$, *para todo* E_{kk}

Es prácticamente imposible describir la relación de modelos que este nivel permite representar ya que dependerá del conjunto de funciones extendidas implementado y dicho conjunto estará supeditado, en este proyecto, a los tipos de ejemplo planteados.

Una vez implementado cada uno de los niveles se presentarán varios tipos de modelos, problemas de distinta naturaleza y ámbito escogidos por el autor, para comprobar la corrección de los niveles existentes y que servirán para explicar adecuadamente las bases teóricas del proyecto así como para ejemplificar las capacidades existentes y futuras.

4 CONOCIMIENTOS PREVIOS

La realización de un proyecto de esta magnitud requiere, tanto del autor como de los posibles usuarios, una serie de conocimientos que podríamos clasificar en tres grupos:

- **Ámbito general**: Aquellos conocimientos en distintas materias que no precisan una explicación, bien por su sencillez, por ser perfectamente conocidos por los lectores o por su escaso impacto en la comprensión del proyecto.

- **Ámbito programático**: Todos los conocimientos propios de la informática y, en concreto, de la programación que se han empleado en el desarrollo de la solución.

- **Ámbito del proyecto**: Los conocimientos particulares al desarrollo del sistema que explican y detallan dicho desarrollo.

Como puede suponerse, la explicación detallada del tercer grupo será objeto de capítulos posteriores dado que se centra en el desarrollo mismo de la solución. Los dos primeros grupos serán comentados en este capítulo en menor o mayor detalle, según la relevancia de cada concepto en este proyecto.

De esta manera el lector dispondrá de la información adecuada para comprender el resto de capítulos del proyecto.

4.1. Conocimientos de ámbito general

La naturaleza del proyecto presentado, un metasimulador capaz de representar muchas y muy variadas situaciones, obliga al conocimiento básico de distintas materias.

Este conocimiento nos posibilita el manejo de la aplicación, su comprensión y la posibilidad de crear nuestros propios modelos.

Los conocimientos requeridos a este nivel serían los siguientes:

- **Matemáticas básicas**: La creación de los modelos requiere conocer los operadores aritmético-lógicos y las funciones matemáticas comunes.

- **Física básica**: Ciertos conocimientos de física, principalmente aquellos sobre mecánica clásica, se hacen imprescindibles en una buena parte de los modelos de simulación.

- **Edición de Textos**: Se debe conocer como editar un archivo de texto plano para poder crear un archivo de configuración que describa nuestro modelo.

- **Normas Common User Access**: Las normas de uso de todo programa bajo el protocolo CUA. Saber abrir, borrar, cargar y ejecutar un programa es sencillo.

- **Autómatas finitos**: Para ciertos tipos de modelos se requiere conocer el concepto de autómata finito, pero solo en estos casos.

- **Programación auxiliar**: Conocer las estructuras con las que, por ejemplo, se gestionan los tics de ejecución, no es relevante para la comprensión del simulador.

- **Lenguaje Java**: El desarrollo del programa se ha realizado con el lenguaje de programación Java de Sun MicroSystems, pero podría haberse hecho en C++ de Borland o en VB.NET de Microsoft Corporation. El lenguaje utilizado no es relevante para la comprensión del proyecto.

- **Compiladores**: En los términos del proyecto es irrelevante conocer la forma en la que un compilador ha pasado líneas en lenguaje java a lenguaje máquina. El interés reside en lo que se ejecuta y en como se ejecuta, no en como se ha obtenido el ejecutable.

Existen multitud de conocimientos particulares, en función de la simulación que deseemos realizar, que no se describen ya que queda en manos del usuario conocer lo necesario para crear un modelo concreto. Cada ejemplo propuesto en este proyecto incluye información suficiente para entender el modelo que se representa.

4.2. Conocimientos de ámbito programático

Este apartado se centra en la descripción de ciertos conocimientos, dentro del ámbito de la programación, que requieren una explicación para poder comprender la solución desarrollada.

Los conocimientos requeridos a este nivel son los siguientes:

- **Estructuras auxiliares**: Aquellas estructuras de datos complejas, suministradas por el compilador, serán brevemente explicadas en cuanto al uso dado.

- **Java 3D**: Se ha empleado la librería gráfica de Java para disponer de un entorno visual de representación por lo que se realizará una breve explicación que describa las cuestiones más relevantes.

- **Java Expression Parser**: Es el motor encargado de leer una expresión matemática y calcular su valor. El cálculo de expresiones pasa por su uso y su uso, por su conocimiento.

Pasemos a detallar cada uno de estos apartados.

4.2.1. Estructuras auxiliares

Es habitual que los peticionarios de un programa indiquen sus requisitos de una forma convencional, como en el siguiente ejemplo:

> *Quiero que cada nueva persona que llegue al aeropuerto pase un primer control y por su tipo de vuelo se le indique la puerta a la que debe trasladarse, donde esperará una cola para dejar su equipaje, luego pasará a un hall conjunto donde volverá a ser enviado a una puerta concreta, por conocer hasta instantes antes de la hora de embarque y a la que irá para hacer cola y pasar al avión.*
>
> *Dentro ya del avión, si vamos delante de un pasajero que se ha equivocado de asiento, debemos retroceder hasta su sitio para que pueda sentarse, Yo y el resto de los que me sigan. Las azafatas cogerán platos de la cocina de arriba a abajo y servirán los menús recorriendo las filas de asientos y en cada fila atendiendo a los asientos de cada una de estas.*

Y así un largo etcétera de situaciones en las que lo real debe ser traducido a un modelo computable.

Para ello se desarrollaron una serie de estructuras básicas que se encargan de almacenar la información de forma concreta y que nosotros debemos escoger en función de cada situación que se nos presente. Así, las pilas de platos, el hall de reunión, las colas de espera y las matrices de asientos entran a formar parte de un programa que tuviera que simular la escena descrita anteriormente.

Como las situaciones pueden ser muy complejas en determinadas ocasiones, se crearon ciertos tipos de estructuras que mejoraran el desarrollo de un programa.

Si a esto unimos el hecho de que las estructuras pasaron a ser objetos, con la estructura y una serie de propiedades y funciones ya escritas que nos ahorraran el trabajo de reinventar la rueda en cada desarrollo, tenemos una riqueza estructural de la que este proyecto ha utilizado parte.

Las estructuras utilizadas por el metasimulador extendido son:

- **Arrays**: Los elementos almacenados ocupan una posición del array de forma que recuperamos el valor de lo guardado preguntando por el elemento en una posición determinada.

  La declaración de un array en java es:

  Tipo Variable[dimensiones];

  donde Tipo es el nombre del tipo de datos utilizado, Variable es el nombre que se le da al array y dimensiones expresa el número de elementos de un array simple, de filas y columnas si es una matriz o de filas, columnas y planos si es un cubo.

Veamos algunos ejemplos de uso:

int vector[10];	*Declaro un vector de 10 números enteros*
double matriz[2,5];	*Declaro una matriz de 2x5 números reales*
String textos[200];	*Declaro un vector de 200 literales*
vector[0] = 3;	*La primera posición de vector vale 3*
textos[2] = "";	*El tercer texto está vacío*
vector[i] = 2;	*La posición i-ésima de vector almacena un 2*
4 + vector[0];	*El resultado de la suma es 7*
textos[2] += "-";	*El tercer texto ahora es un guión*

- **Vectores:** Es un poco engorroso trabajar con arrays ya que son tan sencillos que utilizarlos requiere que el programador programe cualquier operación sobre los mismos.

  Si el programador lo necesita, java dispone de un objeto Vector que incluye la estructura anterior pero, como buen objeto, dispone de propiedades y funciones para su manejo.

  Las diferencias fundamentales entre un vector y un array son:

  - Un vector es un array de **objetos**
  - Un vector puede crecer de forma dinámica

  La declaración de un Vector en java es: *Vector Variable;*

  donde Variable es el nombre dado a la variable de tipo Vector.

  El uso resumido de un objeto Vector es:

Vector v;	*Declaro v de tipo Vector*
v.add(obj);	*Añado elemento obj al final de v*
v.add(i, obj);	*Añado obj en la posición i-ésima de v*
if(v.contains(obj))	*Compruebo si obj existe en v*
obj = v.get(i);	*Cargo en obj el elemento i-ésimo de v*
obj = v.elementAt(i);	*Cargo en obj el elemento i-ésimo de v*
v.remove(i);	*Elimino el i-ésimo elemento del vector*
v.remove(obj);	*Elimino el elemento obj del vector v*
v.clear();	*Elimino todos los elementos del vector v*

- **Pilas**: Es una de las estructuras básicas de cualquier lenguaje de programación, incluido ensamblador, y es equivalente a una pila de platos en una cocina.

  El último plato que se ha puesto en la pila será el primer plato que cojamos para su uso. El nemotécnico anglosajón que se utiliza es **L**ast **I**nput is **F**irst to **O**utput o **LIFO** para abreviar.

  Esta estructura es fundamental en las llamadas a procedimientos de cualquier sistema y no digamos para la ejecución de procedimientos recursivos donde la pila alberga el estado del programa entre llamadas.

  En el caso del proyecto que nos ocupa, se utiliza la pila para almacenar los números de identificador de elementos **Hex** que han sido utilizados alguna vez y que se encuentran disponibles, al haberse eliminado el elemento. De esta forma se intenta evitar que el número de identificador, que se asigna consecutivamente en el rango de los naturales, llegue a valores tan grandes que desborden la capacidad de la variable de tipo entero utilizada para asignar el siguiente número identificador. Antes de asignar un número nuevo, se reutiliza alguno de los de la pila.

  La declaración de una Pila en java es:

  Stack Variable;

  donde Variable es el nombre dado a la variable de tipo Pila.

  El uso de un objeto Pila es:

Stack p;	*Declaro p de tipo Pila*
p.push(obj);	*Inserta el elemento obj en la pila p*
obj = p.pop();	*Extrae de la pila el elemento superior*
if(p.isempty())	*Compruebo si la pila p está vacía*

- **Enumeration**: Como existe una multitud de objetos complejos que podemos utilizar para desarrollar un programa, es muy probable que, en determinados momentos, necesitemos disponer de la lista completa de elementos albergados en la estructura, independientemente de si se encuentran apilados, encolados, ordenados por clave, ordenados por nombre, en árbol o en grafo.

  Para poder disponer de una visión plana de los elementos existentes en alguna de estas estructuras, disponemos del objeto Enumeration que nos permite leer la lista de elementos.

  En nuestro caso, cada vez que se deben examinar las estructuras de almacenamiento de una **Hex** o del conjunto completo de **Hexes**, se recurre a su enumerado para realizar un bucle completo de lectura.

Son tan numerosas las ocasiones en las que se utiliza este objeto en el proyecto que comprender el enumerado hace posible leer el código del mismo.

La declaración de un Enumeration en java es:

Enumeration Variable;

donde Variable es el nombre dado a la variable de tipo Enumeration.

El uso de un objeto Enumeration es:

Enumeration e;	*Declaro e de tipo Enumeration*
obj = e.nextElement();	*Devuelve el siguiente elemento a leer*
if(e.hasMoreElements())	*Comprueba si existen elementos por leer*

- **HashTables**: En el punto anterior se comentó que el objeto de enumeración nos permitía listar los elementos existentes en alguna de las estructuras complejas existentes. Sin lugar a dudas la estructura compleja que se ha utilizado de manera exhaustiva en este proyecto es la de Tabla Hash o HashTable, tal como se la conoce en la mayoría de los lenguajes modernos de programación, java entre ellos.

  Una Tabla Hash no es sino un objeto del tipo array que, a su vez, almacena objetos que pueden ser localizados por su índice o por el valor almacenado.

  Dicho así, parece un objeto vector, pero a diferencia de este, el índice de un elemento puede venir dado por un entero, como en el caso del vector, o por un string o por el objeto que se desee. El índice pasa a ser no un número sino una clave de localización. La otra diferencia principal es que un vector tiene un solo objeto almacenado en una posición, mientras que una Tabla Hash puede almacenar varios objetos en la misma posición. Esa es una de sus principales virtudes.

  Si resumiéramos mucho, podríamos decir que basta con que insertemos un elemento en la Tabla Hash mediante su clave y siempre podremos recuperarlo preguntando por dicha clave. Casi como en una tabla de base de datos. Si la clave primaria de la tabla es simple, como en una tabla de base de datos.

  La palabra Hash del nombre de este objeto viene a significar que, mediante un algoritmo de ordenación basado en funciones de dispersión o funciones Hash, que generan una clave según el valor dado, la estructura coloca cada elemento en posiciones determinadas de la misma y, en caso de que existan objetos con el mismo valor o clave, trata la colisión dada por la repetición generando una lista de valores asignados a la misma clave.
  La potencia es evidente: puedo obtener los datos de una persona, lo que sería un supuesto objeto propietario Persona, con tan solo pasar su DNI, que en este ejemplo haría de clave de la entrada de la HashTable.

La declaración de una HashTable en java es:

Hashtable Variable;

donde Variable es el nombre dado a la variable de tipo HashTable.

El uso resumido de un objeto Hashtable es:

Hashtable h;	*Declaro h de tipo Hashtable*
h.put(key, obj);	*Inserta obj con clave key en h*
obj = h.get(key);	*Obtiene el elemento con clave key*
if(h.contains(obj))	*Comprueba si obj existe en hashtable h*
if(h.containsValue(obj))	*Comprueba si obj existe en hashtable h*
if(h.containsKey(key))	*Comprueba obj por su clave key en h*
Enumeration e; *e = h.elements();*	*Carga en el Enumeration e* *los elementos de h*
if(h.isEmpty())	*Comprueba si h tiene elementos*

4.2.2. Java 3D

La librería Java 3D es un conjunto de clases software escritas en Java para permitir la creación y uso de estructuras gráficas tridimensionales. Esta librería es la encargada de visualizar la ventana de representación gráfica del metasimulador y las respectivas ejecuciones, compuestas de un número variable de prismas hexagonales, creados específicamente para este proyecto.

Si queremos utilizar esta librería en nuestro proyecto, debemos seguir una serie de pasos concretos:

- Crear un Universo Virtual que contenga nuestra escena.
- Crear una estructura de datos que contenga nuestros objetos.
- Añadir los objetos de nuestra elección al grupo.
- Posicionar al espectador de forma que pueda visualizar cada objeto.
- Añadir el grupo de objetos al Universo.

Veamos un ejemplo sencillo con estos pasos:

Traer al proyecto las clases Universo, Grupo y Objeto:

```
import com.sun.j3d.utils.universe.SimpleUniverse;
import javax.media.j3d.BranchGroup;
import com.sun.j3d.utils.geometry.ColorCube;
```

Clase principal del ejemplo con un método inicial y uno principal:

```
public class Hello3d {
        public Hello3d() {
           SimpleUniverse universe = new SimpleUniverse(); //Crear Universo
           BranchGroup group = new BranchGroup(); // Crear Grupo de Objetos
           group.addChild(new ColorCube(0.3)); // Insertar Objeto en Grupo
           universe.getViewingPlatform().setNominalViewingTransform(); // Ver
           universe.addBranchGraph(group); // Añadir el Grupo al Universo
        }

        public static void main( String[] args) {
           new Hello3d();
        }
}
```

La ejecución nos presentará un cubo coloreado. En este proyecto se construyen los hexágonos base, mediante triángulos y con estos y seis rectángulos laterales, un prisma hexagonal. Este prisma es instanciado para cada elemento existente en la simulación de forma que podamos ver la evolución de cualquier instancia del modelo.

4.2.3. Java Expresión Parser

Conocido por su acrónimo **JEP**, esta aplicación es la encargada de entender las fórmulas indicadas en el archivo de configuración y, si dispone de los valores para cada variable empleada en la expresión matemática, devolver el valor de la fórmula escrita.

Si se codifica y compila un programa con dos líneas de entrada solicitando dos números a y b, que devuelva una salida con la suma de a y b, entonces si quisiéramos que sumara un tercer número necesitaríamos avisar al autor del programa para proporcionarnos el código fuente o, esperar que creara una nueva versión que contemplara este tercer número en la suma, o, ……, que el programador dejara la gestión de los cálculos de expresiones a un **parser matemático** que supiera leer los valores de las variables, realizara la operación solicitada y devolviera su valor.

Esta última opción es posible gracias al uso de JEP en el desarrollo de la funcionalidad.

JEP se basa en una estructura de tipo HashTable, comentada en los puntos anteriores de esta sección, que almacena información de todas las expresiones utilizadas, de todas las variables empleadas en alguna de estas expresiones y que, conocidos los valores de dichas variables, es capaz de calcular el resultado de dicha expresión.

Así, para JEP, el ejemplo de la suma sería como sigue:

Declaración del objeto Parser

JEP Parser;

Inicializaciones del Parser

Parser.addStandardConstants();	*Ahora puedo usar π, y e*
Parser.addStandardFunctions();	*Ahora puedo usar Neperianos*
Parser.addComplex();	*Si quiero complejos*
Parser.setAllowAssignment(true);	*Permito asignaciones x = f(x,y,z)*
Parser.setAllowUndeclared(true);	*Permito variables sin declarar*

Carga de sumandos a y b y suma s

Parser.addVariable("a",a);	*Inserto a como 'a'*
Parser.addVariable("b",b);	*Inserto b como 'b'*
Parser.addVariable("s",0.0);	*Inserto variable 's' con valor 0*

Calculo s = a + b

Parser.parseExpression("a+b");	*Sí, aquí es cuando se suman*
suma = Parser.getValueAsObject();	*Guardo en suma el resultado*

Guardo el resultado de la suma

Parser.setVar("s",suma);	*Guardo en 's' el resultado*

4.3. Conocimientos de ámbito exclusivo del simulador

Los conocimientos de este grupo pertenecen por completo al desarrollo del metasimulador presentado por lo que su explicación detallada se realizará en el capítulo dedicado al desarrollo de la solución. En este capítulo solo se hará mención de los distintos conceptos manejados para completar el conjunto de conocimientos que este proyecto precisa.

Los conocimientos a este nivel son los siguientes:

- **Modelo de simulador**: Se necesita explicar la teoría de la representación para saber que requerirá el desarrollo que la soporte.

- **Estructuras del simulador**: Es el modelo de objetos en el que se apoya el simulador para ser lo que es. No hay nada sin esto.

- **Funcionalidad del simulador**: Sabiendo como funciona la simulación podrá comprender el alcance de las simulaciones y la potencia de los modelos.

- **Estructura del archivo de configuración**: Deben conocerse por completo las distintas secciones de un archivo de configuración, los parámetros obligatorios y los opcionales y los valores admitidos para cada parámetro para que el usuario esté en disposición de crear sus propias simulaciones.

- **Funciones extendidas**: En aquellas simulaciones donde las funciones matemáticas no bastarían para poder construir el modelo deseado se deben manejar las funciones especiales creadas para convertir este metasimulador en un metasimulador extendido.

Como se puede adivinar, se ha prescindido de las nociones básicas ya conocidas por los usuarios y el resto de la información que requiera cierta explicación o un detalle preciso se distribuirá a lo largo de los siguientes apartados en los que se verá qué necesita el simulador, cómo se maneja por parte del usuario y los ejemplos realizados para el proyecto.

5 DESARROLLO DEL SISTEMA

Este capítulo trata todo lo referente al metasimulador desde el punto de vista de desarrollo de una aplicación informática. Una vez conocidos los distintos objetivos que deben ser satisfechos, se debe diseñar e implementar una solución software que cumpla los requisitos.

Como sucede en cualquier desarrollo software convencional, se afrontó inicialmente la etapa de análisis del problema mediante la definición del modelo teórico del simulador que se pretendía obtener. Tras esta fase se describieron las distintas estructuras de datos que soportan la solución escogida, así como los distintos módulos funcionales requeridos por el sistema para cumplir todas las exigencias de partida.

5.1. Modelo teórico del simulador

El metasimulador extendido consiste en un programa informático capaz de leer un modelo de simulación dado como entrada y representarlo visualmente a lo largo de la ejecución del mismo.

El modelo a simular, que se pasa como entrada al programa, consiste en un documento en el que se especifican las entidades de simulación que intervienen en el modelo y que son cargadas en el programa para su representación.

Durante la ejecución del modelo se modifican los valores de las variables de cada elemento que disponga de una función asociada basada en funciones matemáticas estándar o en funciones extendidas desarrolladas para la simulación, pueden insertarse nuevos elementos, eliminarse elementos existentes, suspender el cálculo de un elemento o variar su comportamiento según se minimice o no una función objetivo dada.

Una definición más formal del metasimulador sería la siguiente:

Sea MSE el metasimulador extendido

Sea M el conjunto de modelos que MSE puede representar

Sea T el conjunto de instantes de tiempo de ejecución para todo modelo m_i en MSE

Sea F el conjunto de funciones matemáticas estándar para todo modelo m_i en MSE

Sea X el conjunto de funciones extendidas disponibles en MSE

Sea E el conjunto de elementos de un modelo m_i

Sea C el conjunto de Clases de elemento de un modelo m_i

Sea V el conjunto de propiedades/variables de un elemento e_i

Sea F el conjunto de comportamientos/funciones de un elemento e_i

Sea I el conjunto de condiciones de inserción y valores iniciales de un elemento

Sea D el conjunto de condiciones de eliminación de un elemento

Sea S el conjunto de condiciones de parada de un elemento

Sea O el conjunto de funciones objetivo de un elemento

Se cumple:

Todo elemento e_i *de E se representa en MSE como un prisma hexagonal*

$V = VI \cup VE$ *donde*

$VI = \{x, y, z, r, c[]\} / card(VI) = 25$

conjunto de propiedades intrínsecas de un elemento, donde x,y,z representan las coordenadas en el espacio de representación, r representa el radio de un elemento y c[] define los colores RGB del centro y las 6 aristas del prisma hexagonal. Este conjunto debe existir en cualquier modelo m_i *de MSE.*

y

$VE = \{v_1, v_2, .., v_n\} / card(VE) = \{0..N\}$

conjunto de propiedades extrínsecas de un elemento, donde cada variable corresponde a una propiedad definida por el usuario para el elemento y no tiene representación visual en MSE. Este conjunto es opcional para todo m_i *de MSE.*

Todo elemento e_i *de E tiene un conjunto de funciones* $F(e_i) = \{f_1, f_2, .., f_m\}$ *de forma que toda* f_j *de* $F(e_i)$ *cumple que* $f_j = F(e_{j1}.v_{k1}, e_{j2}.v_{k2}, .., e_{ji}.v_{ki}, .., e_{jn}.v_{kn})$ *donde* $e_{ji}.v_{ki}$ *representa el valor de la variable ki-ésima del elemento ji-ésimo del modelo.*

$F = \{f_1, f_2, .., f_m\} \cup \{x_1, x_2, .., x_n\}$ *donde* $\{f_1, f_2, .., f_m\}$ *es el conjunto de funciones matemáticas estándar y* $\{x_1, x_2, .., x_m\}$ *es el conjunto de funciones extendidas de MSE.*

$I = \{I_1, I_2, .., I_n\} / I_i=(CI_i, INI_k)$ *donde*

CI_i *es una expresión* $[or, and, not, e_{i1}.v_{i1}, e_{i2}.v_{i2}, .., e_{in}.v_{in}] / CI_i = \{True, False\}$

INI_k *es una múltiple asignación de valores iniciales* $VI(e_k)$ *y* $VE(e_k)$

Para cualquier elemento e_i *del modelo que cumpla* CI_i *se crea un nuevo elemento* e_k *con valores iniciales de acuerdo a* INI_k

$D = \{D_1, D_2, .., D_n\}$ / $D_i=(CD_i)$ donde

CD_i es una expresión [or, and, not, $e_{i1}.v_{i1}$, $e_{i2}.v_{i2}$, .., $e_{in}.v_{in}$] / $CD_i = \{True, False\}$

Para cualquier elemento e_i del modelo que cumpla CD_i el elemento se elimina del modelo en el siguiente paso de ejecución.

$S = \{S_1, S_2, .., S_n\}$ / $S_i=(CS_i)$ donde

CS_i es una expresión [or, and, not, $e_{i1}.v_{i1}$, $e_{i2}.v_{i2}$, .., $e_{in}.v_{in}$] / $CS_i = \{True, False\}$

Para cualquier elemento e_i del modelo que cumpla CS_i el elemento deja de tenerse en cuenta para el cálculo de valores de variable para los restantes pasos de ejecución.

$O= \{O_1, O_2, .., O_n\}$ donde

O_i es una función objetivo para e_i de la forma $O_i = \varphi * MX_i + [(1 - \varphi) * MN_i]$

$\varphi = 1$, si $O_i(e_i)$ en t_j es mayor que $O_i(e_i)$ en t_{j-1}
$\varphi = 0$, si $O_i(e_i)$ en t_j es menor o igual que $O_i(e_i)$ en t_{j-1}

MX_i y MN_i son funciones aplicables al elemento e_i

El conjunto C se compone de elementos de E, de forma que $Card(C) \leq Card(E)$

Si e_i pertenece a C_i, entonces e_i no pertenece a C_j, para cualquier $j \neq i$

Toda clase C dispone de los conjuntos V, F, I, D, S, O antes mencionados

Todo elemento simple de E perteneciente a una clase C hereda las variables, con sus propios valores, y las Funciones de C

El espacio de representación R^3 puede albergar un número ilimitado de clases C pero solo se visualizarán MaxX*MaxY*MaxZ elementos, de la clase que sea

Sea MSE el Metasimulador Extendido y E_j un elemento simple de la clase C_k y $\{..t_i, t_{i+1},..\}$ el conjunto de pasos de ejecución en el tiempo. Se asegura que todo $E_j(t_i)$ pasará a $E_{j(ti+1)}$ si no se cumple alguna condición D de $E(t_i)$ o alguna condición S de $E(t_i)$.

Así mismo puede ser creado $E_j(t_{i+1})$ si se cumple para algún E_h de la clase C_k alguna de las condiciones I de $Eh(t_i)$, en cuyo caso E_h se dice que es padre de E_j.

$E_j(t_i)$ modifica V_i a V_{i+1} según F o modifica V_i a W_{i+1} según G, si $E_j(t_i)$ no iguala o maximiza la función O de $E_j(t_i)$

Como ya se dijo, el simulador trabaja con abstracciones de datos que representan partículas o grupos de partículas, es decir, un prisma hexagonal puede simbolizar un electrón, un átomo, una molécula, un sistema, un organismo, un mineral, un vegetal, un animal, varios de estos a la vez o ser puntos matemáticos de una curva o elementos de un algoritmo que se quiera ejecutar.

Si hablamos de un electrón, este elemento abstracto podrá tener propiedades tales como su posición o su momento angular, si hablamos de átomos podremos hablar de propiedades como su número de capas, su radio, número de electrones, si se trata de una molécula, hablaremos de sus enlaces entre átomos, si de un sistema, del número de moléculas y su disposición, de un organismo trataremos su velocidad, su masa, de un mineral, su refracción o valor en el mercado, de un vegetal su número de ramas o de si presenta o no flores, de un animal de su orden natural, del número de patas o de antenas o cuantos ojos dispone o de su hambre o de su dinero en el bolsillo y, por concluir, si hablamos de un grupo de personas cruzando la calle, hablaremos de su vector de recorrido, del número de personas, de su velocidad o del tiempo hasta conseguir llegar al otro lado de la calle.

Todos ellos pueden ser representados con un solo prisma hexagonal si el modelo de la simulación, el usuario que lo crea, estipula que esa es la forma más oportuna de visualizarlo. Por supuesto, si modelamos una batalla puede interesarnos que cada elemento simple represente una compañía, de forma que la batalla de Waterloo pueda ser representada con unas decenas de elementos. Si queremos, por el contrario, ver como se mueve un banco de peces cuando un predador intenta comérselos podemos representar N elementos del tipo Pez y un elemento del tipo Predador y simular lo que deseemos al respecto.

Bien, pues todas las propiedades descritas pasarán a ser variables en el sistema y se verán alteradas por las distintas funciones que se puedan aplicar sobre las mismas.

En las líneas anteriores se encuentra descrito el modelo abstracto del proyecto. Como sabemos, existe una implementación del mundo matemático al mundo computacional que deberá tener en cuenta el modelo anterior y soportarlo mediante las estructuras de datos y funciones adecuadas a este.

5.2. Objetos del sistema

Cuando un aprendiz realiza sus primeros pasos en la programación de un lenguaje de alto nivel, pasa ineludiblemente por la escritura, compilación y ejecución del más famoso de los programas de ordenador, *Hello World*.

Salvo este primer ensayo, el resto de programas requerirán que utilice variables para almacenar los datos de entrada y nuevas variables, o algunas de las ya existentes, para mostrar los datos de salida.

Entre la entrada y la salida, debe almacenarse toda la información necesaria para lograr la ejecución en cada paso de todos los elementos que pueden llegar a intervenir en un ejemplo dado.

En líneas muy básicas, el diseño software de la idea teórica del metasimulador exigía esto mismo, así como una funcionalidad que ejecutara lo deseado, apoyándose en estas estructuras.

Trataremos cada uno de estos puntos por separado en los apartados siguientes, empezando ahora por los cajones donde se guarda la información.

Es preciso indicar que java es un lenguaje orientado a objetos y que, por tanto, la mejor manera de pasear por sus calles es utilizando el idioma de la OOP, o programación orientada a objetos, que cambia la visión existente de los programas como colecciones de instrucciones y estructuras de datos a una visión en la que creamos o reutilizamos entidades que encapsulan unos datos y las funciones sobre los mismos.

De esta forma es más propio hablar de un gestor de clientes, o de memoria del sistema o un gestor del espacio de simulación, como es nuestro caso, en donde se han agrupado los datos y las propiedades junto con las funciones mediante las cuales se gestionan o se comunican sus valores, así como los eventos sobre dicha entidad.

Por ello disponemos de una colección de clases que pertenecerán a uno de los siguientes tipos:

- **Clases de información**: Se necesita explicar la teoría de la representación para saber que requerirá el desarrollo que la soporte.

- **Clases de datos**: Datos, datos y más datos para el simulador. Se entiende por datos, las distintas estructuras de datos que pone a disposición del sistema.

- **Clases funcionales**: Algún dato o ninguno y casi todo funcionalidad. Estas clases hacen cosas con los datos de otras.

- **Clases mixtas**: Un gestor que necesita sus datos y tiene sus funciones para los mismos. Una clase balanceada, más típica en OOP que suele ser usada para los gestores de áreas concretas del simulador.

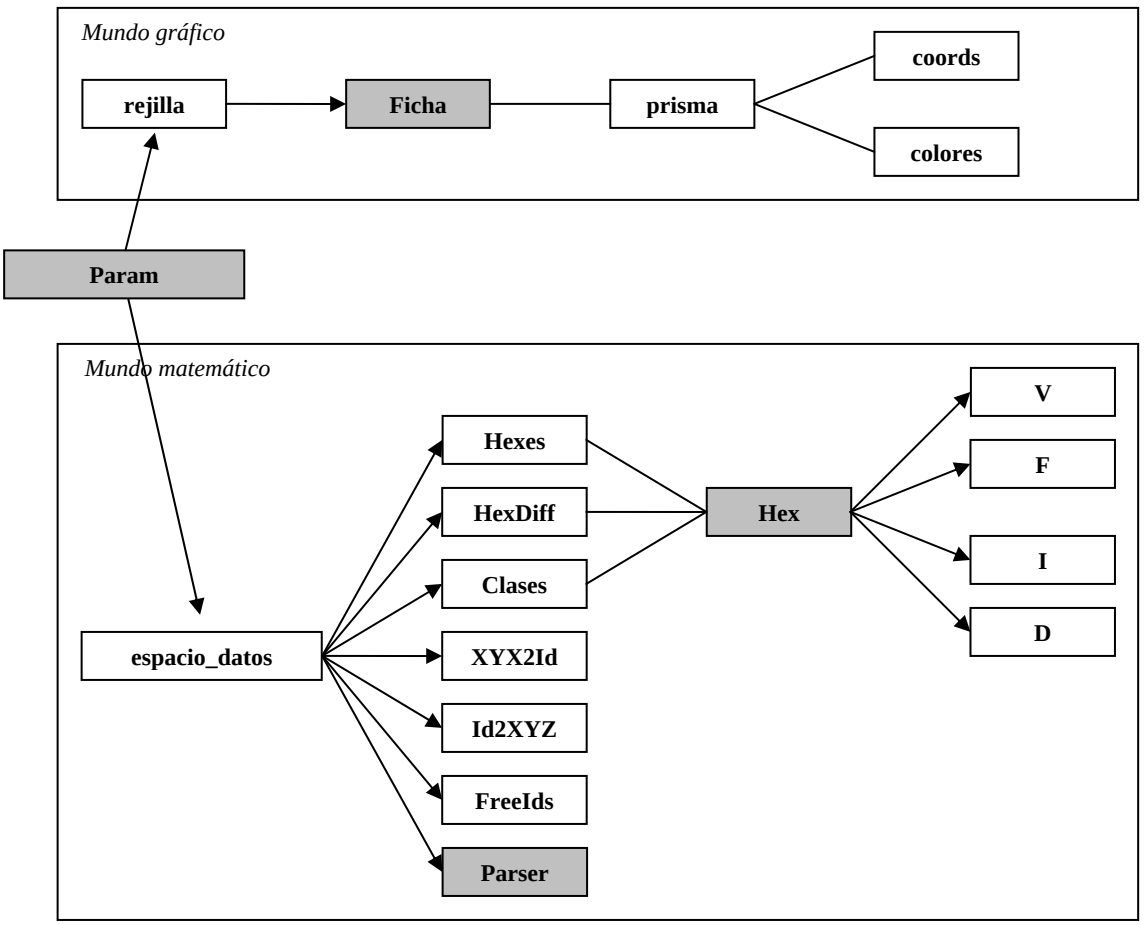
5.2.1. Mapa global de objetos

Es mejor, antes de comenzar todas las descripciones necesarias, que el lector vea un pequeño mapa resumen con las estructuras mas relevantes.

Si bien los objetos son conjuntos de estructuras de datos y de funciones que realizan determinadas tareas sobre estos datos y los de aquellos objetos a los que tengan acceso, nos centramos ahora en una visión general de las estructuras, aquellas que han sido diseñadas con el único objetivo de soportar los distintos atributos realtivos al metasimulador y hacer posible su ejecución.

Mas adelante observaremos la funcionalidad de los objetos diseñados, que emplean estos mismos datos.

Esta sería la foto de familia del metasimulador extendido:



Como puede apreciarse, todo es obtenido de la clase **Param** en donde se han reunido aquellas estructuras propietarias que se requieren para poder gestionar toda la información necesaria.

Tras esta clase se crean el resto de estructuras, divididas en dos grandes mundos, el gráfico y el matemático.

El metasimulador mantiene los estéticos prismas de color y los grisáceos hex separados en todo momento, de forma que con unos vemos que ocurre en la simulación y con los otros sabremos el valor exacto de cada una de las propiedades escritas. Recuerde el lector que un prisma se encontrará en (x,y,z), siendo x,y,z números naturales, pero su hex tendrá como coordenadas (u,v,w), siendo u,v,w números reales.

Cuando se necesite comunicar ambos mundos, lo haremos a través de un par de arrays que realizan la función de traducción Ficha-Hex y Hex-Ficha, como ocurre con los diccionarios.

En resumen, **Param** instancia una **rejilla** gráfica de datos y un espacio, **espacio_datos**, matemático.

La rejilla no es sino un cubo de elementos **Ficha** que representan cada **prisma** hexagonal en la pantalla de visualización, para lo que se necesita un TriangleArray standard de java3D y dos arrays mas, uno, **coords**, para las distintas coordenadas de los triángulos que conforman los hexágonos del prisma y otro para los **colores** del centro y las distintas aristas de estos.

De forma paralela, dada la relación entre los dos mundos, se crea desde el espacio matemático un conjunto de **Hexes** en el espacio, un conjunto **HexDiff** con las diferencias entre pares de ejecuciones consecutivas y un conjunto que contabiliza las distintas **Clases** existentes en la simulación actual. Los tres grupos, lo son de un solo tipo de clase, nuestra ya famosa clase que define el elemento matemático llamado **Hex** y que contiene el conjunto de **V**ariables, las **F**unciones que representan cada ecuación de un elemento, las condiciones y ecuaciones de **I**nserción y las condiciones de **D**elete que pueden eliminar un elemento de la simulación.

Para poder comunicar lo gráfico y lo matemático, tenemos las dos estructuras de traducción ya comentadas, una para saber que Id tiene el elemento en posición XYZ, llamado **XYZ2Id** y otra para saber que coordenadas (X,Y,Z) tiene un elemento con Id concreto, llamado **Id2XYZ**, como no podía ser de otra forma.

Dado que se intenta reutilizar Ids empleados en elementos eliminados de la ejecución, la estructura **FreeIds** mantiene una pila con estos valores y, por último se instancia una variable **Parser** para disponer de un parser de fórmulas matemáticas.

Ahora que disponemos de la relación de estructuras principales, comenzaremos su descripción en mayor detalle.

5.2.2. Objetos principales del simulador

Dado que se han creado clases de objetos y estructuras basadas en tipos estándar del compilador java sobre estos, dejaremos para la parte funcional el uso de todos ellos y en este apartado nos centraremos en los objetos propietarios puros, es decir, aquellos que ha sido necesario crear desde elementos simples, objetos java o combinaciones de los mismos que dan forma computacional a la idea del metasimulador, obviando aquellos objetos que, siendo importantes, son auxiliares del concepto fundamental.

Por poner ejemplos aclarativos, podemos decir que Hexes, la colección de elementos Hex, es importante en el simulador pero es Hex el ladrillo constructivo y Hexes una mera colección de estos ladrillos.

Dado que Hex representa la idea del elemento matemático en la simulación, nos centraremos en dicho objeto y mencionaremos a Hexes, como atributo de la clase Espacio, que representa el simulador en términos matemáticos.

De igual forma, la clase Ficha es el ladrillo constructivo en la parte gráfica y rejilla no es sino un cubo de componentes Ficha, que representan gráficamente el espacio visual del simulador, pero que no aportan mayor conocimiento, salvo el hecho de saber que lo visto, rejilla, se compone de lo esencial, la ficha.

Por todo esto, el siguiente apartado tan solo describirá en detalle las estructuras propietarias fundamentales y dejará la visión de todo el conjunto cuando sus elementos empiecen a bailar los compases de la aplicación en la parte funcional del metasimulador.

5.2.2.1. Objeto Hex

Tal como su nombre nos indica, esta clase implementa un elemento simple del simulador. Básicamente contiene los atributos necesarios para albergar la información de cada elemento.

Un objeto de la clase Hex, o un Hex, contiene lo siguiente:

- Propiedades simples:

  Id Literal con un número natural que identifica inequívocamente este elemento del resto.

  Class Literal con el nombre de la Clase de Elemento a la que pertenece el objeto Hex

  Parent Literal con el Id de la Hex padre de esta

  X, Y, Z Componente X,Y,Z del vector de posición en R3

  R Radio del Hex

  Colors Matriz 7x3 con los valores RGB de color para cada una de las 7 posiciones del hexágono visualizado. Un color para el centro y 6 para las aristas.

  Status Estado del elemento para el siguiente paso de ejecución que indica si no ha variado, si debe ser eliminado, si acaba de ser creado, si ha modificado sus colores, si se ha movido de posición o si ha sido modificado en alguna de sus propiedades.

  Computed Valor booleano que nos indica si el Hex ha sido tratado o no para el siguiente paso de ejecución por el sistema.

  prevX Valor de X en el paso anterior

  prevY Valor de Y en el paso anterior

prevZ Valor de Z en el paso anterior

Stop Literal con la expresión condicional de parada

Obj Literal con la función objetivo del Hex

- Propiedades complejas:

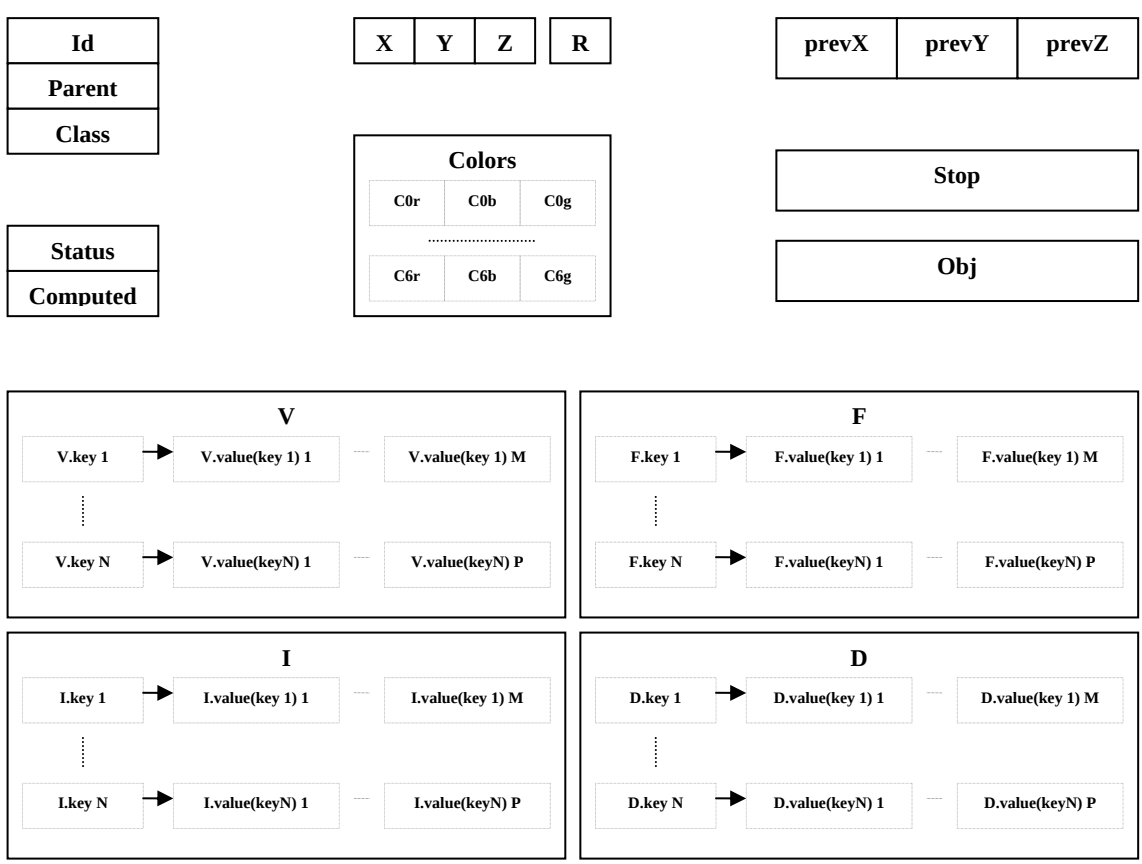
V HashTable con las variables del objeto Hex

F HashTable con las funciones del objeto Hex

I HashTable con las expresiones de inserción de un Hex

D HashTable con las condiciones de supresión de un Hex

Un resumen esquemático de esta clase quedaría como sigue:



5.2.2.2. Objeto Espacio

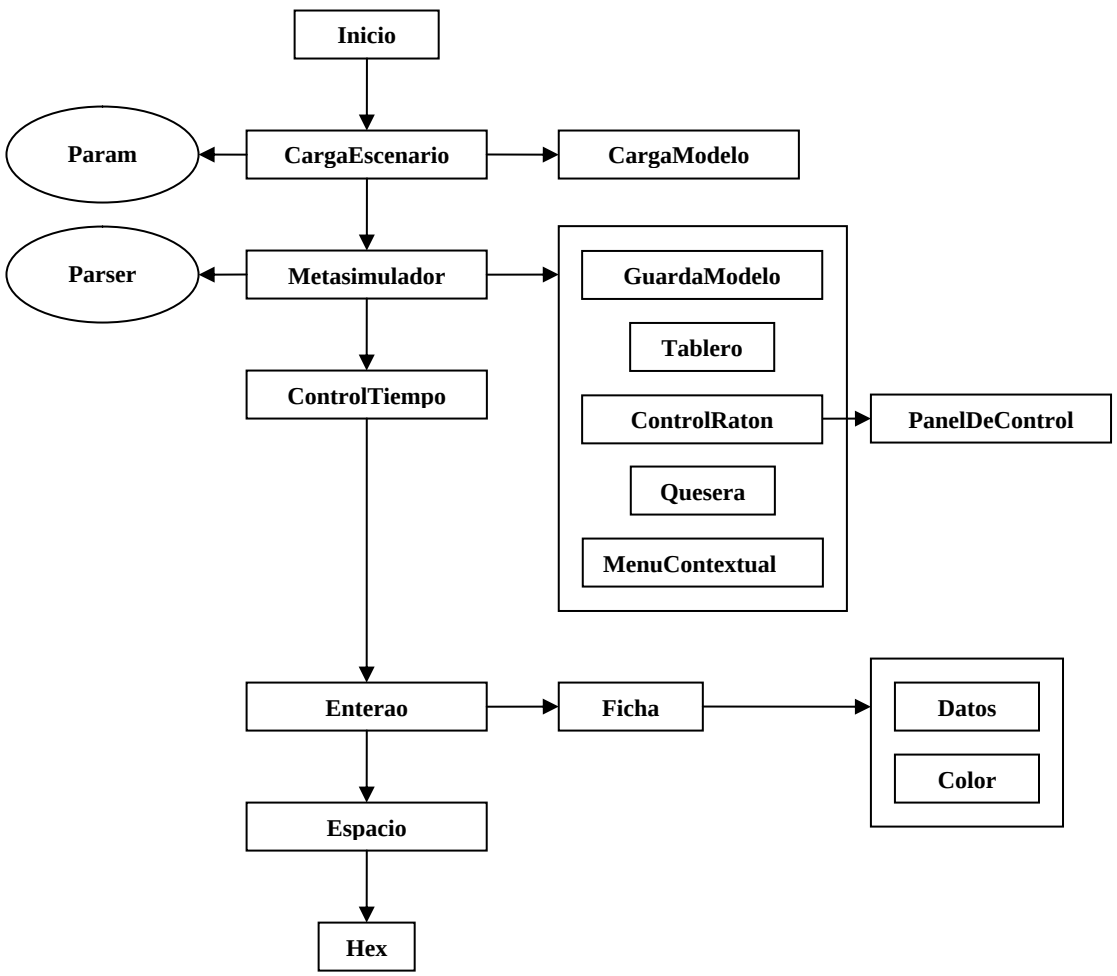
Todos los objetos Hex se agrupan en un espacio del modelo que se define con la siguiente clase.

- Propiedades:

Hexes	Hashtable con todos los Hex existentes
HexDiff	Hashtable con las diferencias respecto a Hexes
Classes	Hashtable con los Hex simbólicos de cada Clase
OrderF	Array con el orden de ejecución de las funciones
Global	Hex con los datos globales a la simulación
XYZ2Id	Matriz 3x3x3 con el Id del Hex en posición (X,Y,Z)
Id2XYZ	Array con la posición (X,Y,X) de un Hex con Id dado
nextId	Entero con el Id a asignar a un nuevo Hex
FreeIds	Stack con los Id reutilizables de Hex eliminados
Parser	Objeto JEP para el cálculo de expresiones del modelo

5.3. Funcionalidad del simulador

Como en los créditos de las películas, veamos un diagrama de objetos del proyecto por orden de aparición.



Si bien, en leguajes orientados a objetos, todo se representa mediante clases, algunas de estas son estructuras de datos con su debida funcionalidad y otras son solo funcionalidad. Los cuadros son las distintas clases del proyecto que realizan labores funcionales en la aplicación y los círculos son las clases para las estructuras de datos más destacadas del proyecto, de las cuales ya hemos hablado en apartados anteriores.

Supongo que una buena forma de describir la funcionalidad subyacente es contar la película, que ya ha comenzado por sus títulos de crédito.

Cuando el usuario solicita al sistema operativo que ejecute el metasimulador extendido, este programa inicia su código por las instrucciones existentes en su clase inicial, llamada como no podía ser de otra forma, **Inicio**.

Inicio tan solo hace una cosa, en términos de resumen de las principales secciones funcionales, que es generar una instancia de la clase **Escenario** la cual solicita al usuario un modelo a simular, de entre los archivos de configuración existentes en el directorio de modelos de la aplicación. Una vez seleccionado el archivo de configuración, el programa instancia la clase **CargaEscenario** que tiene dos cometidos principales.

El primero de ellos consiste en crear una instancia de la clase **CargaModelo** que empezará a leer cada una de las líneas de las que conste el archivo de configuración, detectando el inicio de cada una de las secciones que lo componen y generando las pertinentes estructuras de almacenamiento de la información del modelo a medida que las descubre en el archivo.

El segundo de ellos, si la lectura del mismo se completa sin errores, consiste en preparar el entorno visual 3D mediante la llamada a una instancia de la clase **Metasimulador**. Esta clase carga el lienzo 3D, jerga de java3D, añade el menú del universo visual, el menú de archivo y el panel de mensajes de la aplicación.

Tras esto, crea el grafo de escena del universo visual y lo adjunta a la instancia del universo que se está creando. A partir de aquí se dan las oportunas órdenes a java3D para que cree un universo simple y lo muestre por pantalla, junto al resto de paneles de la aplicación.

Con un universo java3D listo, tan solo falta proporcionarle unas dimensiones adecuadas a nuestra resolución de pantalla y cerrar el menú de selección del archivo de configuración.

En este punto, si hemos escrito en el archivo de configuración que nuestro modelo tenga de partida una ficha azul marino en el centro del paralelepípedo recto rectangular que delimita nuestro universo de prismas hexagonales, veremos una ventana de fondo negro con un bonito paralelepípedo recto rectangular, sus aristas de color blanco, en un inmenso fondo negro y en el centro geométrico de este espacio podremos observar un pequeño punto, o un flamante prisma hexagonal dependiendo del zoom indicado en el archivo de configuración, de un precioso color azul marino.

Esto funciona y ya solo queda que botemos el barco de la simulación apretando el botón con título *Iniciar*. Empieza lo bueno.

En la clase Metasimulador, tras la preparación del universo gráfico, y entre la creación de una instancia de la clase **Tablero**, que visualiza nuestro espacio de representación de aristas blancas inicialmente plano, la creación de una instancia de la clase **ControlRaton**, que nos permite utilizar nuestro dispositivo de ratón como un joystick de navegación visual y la creación de una instancia **Quesera** que pone tres dimensiones a nuestro, de momento, tablero plano de representación, se ejecuta la creación de una instancia **ControlTiempo** que es quien genera los tics del típico reloj sin fin que arbitra nuestra ejecución desde ese mismo momento.
Una vez se ha pulsado el botón *Iniciar* del metasimulador, comienza el reloj a ponerse en funcionamiento y tan solo ejecutará los procedimientos de la instancia de la clase **Enterao** cuyos nombres determinan a la perfección su comportamiento: *ModificaDatos()* y *RepresentaDatos()*.

Este sería el resumen más breve que se podría hacer de la funcionalidad de esta aplicación. El metasimulador extendido ejecuta de forma inicialmente indefinida el par de acciones que calculan los valores del modelo a simular para un nuevo paso y los representa en pantalla.

Y vuelve a calcularlos para el paso siguiente y los vuelve a representar y los calcula y los representa y así hasta que la simulación falle, la ejecución falle o el usuario decida parar el programa o solicitar una nueva simulación.

Es probable que el lector haya deducido que un programa que solo calcula los datos para un nuevo paso y los representa posteriormente o es un programa muy pequeño o encierra en el cálculo y en la representación todo el core funcional. Casi hubiera acertado.

Realmente encierra todo su potencial en el cálculo matemático de los nuevos valores de los elementos. Veamos este potencial.

Cuando se ejecuta el método ***ModificaDatos*()** de la instancia de la clase *Enterao* se llama a un único y poderoso método de la instancia creada de la clase **Espacio** con un nombre muy comercial.

El método ejecutado es ***doIt()*** y su nombre es su declaración de intenciones. Este método lo hace. Todo.

El método *doIt()* no es sino una colección de bucles que recorren toda la estructura de elementos de la siguiente forma:

- Ejecutar el procedimiento ***resetDifferences()***, que recorre todos los elementos de la estructura Hexes, reseteando cada Hex existente para que el elemento se encuentre en un estado inicial de cálculo. De esta forma reseteamos el estado de los elementos, limpiando la estructura de diferencias entre elementos Hex, llamada HexDiff, y poniendo la propiedad Status de cada Hex a *unchanged*. Todo queda listo para un nuevo cálculo de valores.

- Calcular los nuevos valores de las funciones globales, de forma que antes de calcular la nueva posición de nuestra entrañable manzana cayendo del árbol, echemos un vistazo a la propiedad global de la gravedad por si existe alguna función que un endiablado usuario ha querido establecer para no mantener constante la aceleración en su tierra simulada. Si el sol calienta con menor intensidad en función de la hora o si el volumen del océano se reparte en una superficie mayor en momentos de luna llena, son cuestiones que afectarán a todos los elementos de la simulación, por lo que obtendremos los nuevos valores antes de que estos deban ser usados en aquellos Hex que lo mencionen en sus ecuaciones.

- Calcular los nuevos valores para cada elemento Hex, que pasa por tratar:

    - Condiciones de parada: Si existen y se cumple alguna de estas condiciones, la ficha se mantiene en la representación visual pero no se vuelven a calcular sus valores.

    - Condiciones de borrado: Si la ficha no se ha parado en el paso anterior y si existen y se cumple alguna de estas condiciones, la ficha es eliminada, tanto visual como matemáticamente.

    - Condiciones de inserción: Si la ficha no se ha parado ni se ha eliminado, entonces si existen y se cumple alguna de las condiciones se crea una nueva ficha, siguiendo las instrucciones que figurasen a continuación de la condición de inserción que se ha cumplido.

    - Funciones de cálculo: Si la ficha no se ha parado ni se ha eliminado de la simulación, entonces se ejecutan de forma ordenada cada una de las ecuaciones que se almacenan en la hashtable F de este elemento. Cada ecuación requiere variables propietarias de este elemento, variables pertenecientes a otro elemento y aquellas variables globales que se incluyan en la expresión de cálculo para la ecuación dada. Conocidos todos los valores de las variables se solicita al parser que devuelva el valor resultante de la ejecución de esta función y se guarda el resultado en la variable adecuada de V para este elemento.

Una vez ejecutados los procesos anteriores en todos los elementos existentes, nos encontraremos un modelo que podrá tener Hex paradas, Hex eliminadas, nuevas Hex insertadas y Hex con nuevos valores para algunas propiedades o para todas ellas, así como nuevos valores globales. Solo queda representarlo en pantalla para que el usuario vea el resultado en ese paso.

6 DECLARACIÓN DE MODELOS

Tal como se ha comentado en capítulos anteriores, existen multitud de simulaciones que un usuario puede realizar pero para todas ellas se necesita aprender a hablar el pequeño lenguaje de definición de modelos que se ha desarrollado para que la aplicación responda con la representación que deseamos.

En este capítulo se describirá la estructura de un fichero de configuración, tanto las partes generales, las secciones de cada parte y cada uno de los apartados existentes en dichas secciones, de forma que un usuario se encuentre en disposición de introducir sus propios modelos de prueba.

Supongamos que queremos simular la caída de una manzana desde 120 prismas de altura. El modelo se representaría mediante el archivo de configuración siguiente:

```
# Modelo: Caída de una manzana
 [[PRIMERA_PARTE]]
        [ESCENARIO]
                POS_Z=-6.0
                N_HORIZ=204
                N_VERT=402
                N_NIVELES=20
                CIRCULAR=TRUE
                N_MS=10
        [VENTANA]
                DIM_X=1250
                DIM_Y=950

 [[SEGUNDA_PARTE]]
        [GLOBAL_VAR]
                gravedad=9.8
                Tiempo=0
        [GLOBAL_FUNC]
                Tiempo=Tiempo+1
        [CLASE_DEF]
                Manzana
        [CLASE_VAR]
        [CLASE_FUNC]
                Manzana.Y=Manzana.Y-(gravedad*sqr(Tiempo))
        [CLASE_INS]
        [CLASE_SUPR]

[[TERCERA_PARTE]]
        [FICHA_DEF]
                0 100 120 10 1 Manzana
        [FICHA_COLOR]
                0 1 2 255 0 0  0 0 255
        [FICHA_VAR]
        [FICHA_FUNC]

[[CUARTA_PARTE]]
        [CLASE_PARADA]
        [CLASE_CAMBIO]
        [CLASE_OBJ]
```

Evidentemente no está todo ni puede verse todo el potencial, pero ya se irán simulando modelos mas complejos y/o mas ricos en términos de condiciones.

Dado que el objetivo del proyecto no es el de establecer un potente y flexible método de adquisición de modelos, se ha decidido recurrir al modesto pero potente archivo de configuración típico de los sistemas operativos no gráficos, de forma que en un documento de texto plano pueda escribirse sin muchas complicaciones lo que el autor desee simular.

Así, se tendrán dos tipos de líneas de información. Las líneas en blanco o marcadas en su primera columna con un carácter **#** se consideran no parseables por el sistema, es decir, como si no existieran para el metasimulador, ya que solo están a efectos de legibilidad las primeras y de legibilidad y comentarios las segundas. El otro tipo de líneas son las que serán leídas por la aplicación para conocer el modelo que el usuario desea simular.

Si miramos el ejemplo veremos que consta de cuatro partes, identificadas con dobles corchetes, de las que tres son comunes a todo metasimulador y una cuarta parte solo existe en el metasimulador extendido. Cada una de las partes tiene una serie de secciones que se identifican con corchetes simples. Todas y cada una de las partes y secciones deben ser escritas, aunque se encuentren vacías de contenido.

Si no hemos cometido ningún error de escritura en nuestro archivo, podremos guardar este archivo con el nombre manzana y la extensión **conf**, que es la que se utiliza como extensión de los modelos del metasimulador y ejecutar dicho modelo haciendo que el metasimulador lo lea y simule.

Ahora veamos en detalle las cuatro partes y las oportunas secciones de un fichero de configuración que nos servirán para definir cualquier modelo, aprendamos lo que es obligatorio, lo que es prescindible, los valores posibles para cada parámetro de la simulación y el juego de variables, funciones y condiciones que podemos llegar a escribir en este fichero.

6.1. Primera parte. Parámetros generales

El primer apartado es donde podremos configurar como se comportará el metasimulador. Una serie de parámetros que se han determinado de utilidad para el usuario son modificables por este para ajustar la simulación a sus deseos o a las exigencias de representación visual que cada modelo tenga.

Este apartado tiene dos secciones fijas y cada sección tiene una serie de líneas también fijas.

Es posible que un usuario dedicado a pruebas de un tipo concreto de simulación no necesite tocar nada de esta zona pero terminará por probar distintos tipos que requerirán una configuración distinta.

Las secciones son las siguientes:

6.1.1. Sección Escenario

En esta sección se permite que el usuario configure el espacio de simulación mediante la modificación de los valores de los siguientes parámetros:

- **POS_Z**: Posición del espacio de representación sobre el eje Z. Para entendernos mejor, es el zoom de la representación. Si piensa en usted dentro del universo de java3D, usted estaría en la posición 2.4 del eje Z y dando algún valor del rango permitido usted alejaría o acercaría el modelo o usted se acercaría o alejaría al modelo, lo que prefiera pensar que sucede.

- **N_HORIZ**: Número de líneas horizontales de la representación. En el ejemplo anterior se indican 204 líneas, lo que hará que una ficha pueda estar en cualquiera de los 205 valores X visuales, del 0 al 204.

- **N_VERT**: Número de líneas verticales de la representación. En el ejemplo anterior se indican 402 líneas, lo que hará que una ficha pueda estar en cualquiera de los 403 valores Y visuales, del 0 al 402.

- **N_NIVELES**: Número de líneas de altura de la representación. En el ejemplo anterior se indican 20 líneas, lo que hará que una ficha pueda estar en cualquiera de los 21 valores Z visuales, del 0 al 20.

- **CIRCULAR**: Indicamos si se desea que el espacio se comporte como una esfera, de forma que las posiciones fuera del rango de representación vuelvan a dicho rango. Si recuerdan los pasillos laterales del juego de los comecocos, es exactamente lo mismo. En el ejemplo inicial, cuando la manzana llegue al suelo, en Y=0, seguirá cayendo desde Y=402 de nuevo hacia la tierra.

- **N_MS**: Número de milisegundos entre paso de ejecución. Si un modelo de átomos es demasiado rápido, basta aumentar este valor hasta que se comporten con la cinemática propia de unas bolas de billar.

6.1.2. Sección Ventana

En esta sección se permite que el usuario configure la ventana gráfica que nos muestra el espacio de simulación. Como sucede en cualquier ventana de windows, serán únicamente dos parámetros los que se podrán tratar:

- **DIM_X**: Número de píxeles en el eje horizontal de la ventana gráfica.

- **DIM_Y**: Número de píxeles en el eje vertical de la ventana gráfica.

6.2. Segunda parte. Secciones de Coeficientes, Globales y de clase

El segundo apartado comienza a describir la simulación en sí, tratando todo lo referente a las variables y funciones globales, como la gravedad o el tiempo en el ejemplo que manejamos, pero pudiendo insertarse todo aquello que se necesite para el modelo concreto a ejecutar.

A su vez permitirá que creemos las clases necesarias en nuestra simulación, de forma que todos los elementos de cada clase definida hereden las variables y funciones de sus clases.

Aparte de todo lo anterior, el programa nos permite declarar coeficientes de ejecución que permitirán al usuario variar su valor durante la ejecución del modelo dentro de un rango declarado.

Este apartado tiene siete secciones fijas que son las siguientes:

6.2.1. Sección Coef_Var

Podemos, si así lo deseamos, establecer tres coeficientes variables de forma que el simulador nos mostrará, para cada uno de ellos, una barra de desplazamiento que se moverá entre el rango mínimo y máximo declarado para el coeficiente. Las funciones que usen estos coeficientes en su expresión pueden modificar sus valores acorde a la variación de la barra de desplazamiento del coeficiente movida por el usuario mientras la aplicación simula el modelo.

6.2.2. Sección Global_Var

Cada variable global que necesitemos en la simulación se declarará en una línea de esta sección, igualándose a un valor concreto. En nuestro ejemplo hemos declarado las variables *gravedad* de la tierra y T*iempo* de la simulación con valores de 9.8 m/seg^2 y 0 segundos respectivamente.

6.2.3. Sección Global_Func

Cada variable global que necesitemos modificar en cada paso de ejecución, de acuerdo a una función lineal concreta requerirá que escribamos aquí la ecuación que la modifique. En nuestro ejemplo la gravedad nunca se ve alterada por lo que no necesita declararse una función para esta, pero el tiempo deberá aumentar de uno en otro segundo en cada paso de ejecución, por lo que escribiremos *Tiempo=Tiempo+1* para que la variable Tiempo incremente su valor en un segundo por cada tic de ejecución.

6.2.4. Sección Clase_Def

Si necesitamos crear una clase de elementos para el modelo, tan solo bastará con escribir aquí el nombre que llevará la clase. Cuando el metasimulador lea una de estas líneas, preparará lo necesario para tener en cuenta la clase.

En nuestro ejemplo hemos escrito *Manzana* como nombre de la clase de elementos Manzana, aunque solo utilicemos una manzana. Si quisiéramos ver caer doscientas manzanas sería lo mismo, una sola línea con el nombre Manzana escrito en esta sección.

6.2.5. Sección Clase_Var

Una vez hemos declarado alguna clase en la sección que acabamos de comentar, es posible que necesitemos utilizar alguna variable que no sea intrínseca, recordemos que X,Y,Z,R y los 7 colores son intrínsecas para el sistema, por lo que será aquí donde se declararán todas las variables extrínsecas de todas las clases existentes. La declaración se hará, al igual que con las variables globales, mediante una asignación de la variable declarada a un cierto valor o expresión.

6.2.6. Sección Clase_Func

Si tenemos alguna variable de clase que modifique su valor en cada paso de ejecución de acuerdo a una función dada, pondremos aquí la ecuación necesaria para ello. El resto de variables, como en el caso de las funciones globales, no necesitarán escribir aquí una función.

Dado que estamos simulando la caída de una manzana, dicha caída supone que el elemento que representa esta fruta pierda magnitud en su variable Y tal como la gravedad impone, a razón de la expresión

*Manzana.Y=Manzana.Y-(gravedad*sqr(Tiempo))*

por lo que escribimos esta ecuación para que el simulador la tenga en cuenta para cualquier manzana que deseemos dejar caer al suelo.

6.2.7. Sección Clase_Ins

En cualquiera de las clases existentes, podremos decidir condiciones de inserción por las que, si se cumplen, sea creada una nueva ficha de la clase. Supongamos que cuando la manzana llega al suelo nos apiadamos del árbol del que cayó y le ponemos otra manzana en la rama. Para ello deberemos escribir la siguiente orden

(Manzana.Y==0) Manzana.X=100 Manzana.Y=200 Manzana.Z=10

que viene a significar que cuando la manzana llega al suelo, su variable Y vale cero, entonces el simulador deberá crear una segunda manzana en (100, 200, 10), otra vez en la rama del árbol, y esta manzana ya será tratada por el programa de forma conveniente. Dicho de otra manera, tendremos a dos manzanas cayendo al suelo.

6.2.8. Sección Clase_Supr

Si deseamos que alguno de los elementos de una clase, que se simulan en ese momento, sea eliminado si se cumple una condición, escribiremos aquí la condición a cumplir.

Por ejemplo, si queremos que la manzana sea atomizada al contactar con el suelo, bastará con que escribamos en este apartado la siguiente expresión

(Manzana.Y==0)

cada manzana que llegue la suelo, ¡puf!, desaparecerá sin alardes y para siempre. Esto dejará a la manzana insertada en el punto anterior sola y cayendo al suelo mientras se preguntará donde ha ido a parar su compañera hasta que, puf, también desaparezca. Este tipo de desapariciones deberán ser controladas por el usuario si quiere mantener a su público observando alguna manzana. Pero mas adelante veremos cómo mejorar tanta atomización.

6.3. Tercera parte. Secciones de fichas

La tercera sección es, en una parte, mas una deferencia para el usuario que algo absolutamente necesario para una simulación. Cuando se definió el simulador se pensó que la persona que decidiera escribir un modelo podría querer definir un trondentosilobite, que es algo que no existe que se sepa, en vez de una vulgar manzana y como cosa única no necesitaría pasar por la declaración de una clase para utilizar un solo elemento de la misma.

Siendo así, el usuario puede escribir en esta sección lo necesario para definir una ficha o varias fichas, sin ninguna relación entre si, ni pertenecientes a ninguna clase u orden y realizar las simulaciones pertinentes.

Se indica que no es necesaria porque la matemática enseña que un elemento es un conjunto de cardinalidad 1, pero es un conjunto, por lo que puede representarse una sola ficha atendiendo a su clase de un solo elemento instanciado que es lo que se ha hecho en el ejemplo de la manzana que estamos viendo.

Esto rige únicamente para la parte comentada de esta sección. La otra parte es vital. Sin las declaraciones realizadas aquí no hay instancias de la clase manzana ni de nada.

Pero si los amantes de los trondentosilobites quieren, esto es lo que deberán cumplimentar en este apartado:

6.3.1. Sección Ficha_Def

Toca definir aquí cada elemento singular que necesitemos, por lo que el único espécimen de trondentosilobite, o un buen cesto de manzanas, deberán ser declarados de la siguiente forma:

0 100 120 10 1 Manzana
1 276 382 12 1 -

Seguro que ven la definición de la ficha de la clase manzana pero se deben estar preguntando si el tronden ….., como se diga, es la segunda línea. Si, si que lo es. Por eso su clase es **–** que significa para el simulador que esa ficha no pertenece a ninguna clase.

Entonces ¿como sabemos que ese elemento es lo que es? Muy sencillo. ¿Les suena la frase de "*por sus actos los conoceréis*"?, pues por los mismos se sabrá y no de otra forma. Esos actos serán las variables y, sobre todo, las funciones para esas variables que se indiquen mas adelante en esta sección. La manzana, como ya se ha comentado, tiene inmunidad diplomática y puede prescindir de declarar lo anterior. Le basta con decir que viene con el grupo Manzana para que la dejen pasar. Por eso la ficha 0-ésima incluye el nombre de una clase declarada en vez de un guión.

6.3.2. Sección Ficha_Color

Y si queremos que una manzana exhiba un rojo tentador, de esos que hacen que quieras darle un bocado, este es el momento de exhibir tales galas. Para ello el usuario deberá escribir la siguiente línea

0 1 2 255 0 0 0 0 255

que significa que la ficha 0-ésima, nuestra manzana, tiene un solo color y una modalidad de pintado basado en un degradado del color del centro hasta las aristas del prisma hexagonal. El color del centro lo pintamos en un RGB 255, 0, 0 y el de cada arista lo pintamos en un RGB 0, 0, 255 lo que significa que el centro, de color rojo puro, se degrada hasta alcanzar un color azul puro en sus aristas.

Imagino que querrán hacer sus propias gamas de colores para los distintos modelos que deseen realizar con el metasimulador por lo que explicaré brevemente cada uno de los códigos que se utilizan.

Para ello, definamos la sintaxis estricta de la línea de la forma

IdFicha Colores TipoCentro R_0 G_0 B_0 R_1 G_1 B_1 *[*R_2 G_2 B_2 .. R_6 G_6 B_6*]*

donde:

- **IdFicha** es el número de ficha de la representación. Los colores definidos a continuación y su uso se aplicarán solo a la ficha con este índice.

- **Colores** es el número de colores que tendrá la ficha. Este valor puede ser cualquier número natural de 1 a 6.

- **TipoCentro** es un número que define la forma de aplicación del color a la ficha dada, teniendo como referencia el centro. Si el valor es 1 entonces los colores de las aristas llegan sin uso de degradado hasta el centro del hexágono. Si el valor es un 2, entonces los colores de las aristas se degradan desde su color hasta el del centro. Si en cambio utilizamos el 3 se conseguirá como color del centro una mezcla de colores de las aristas basado en el color medio de las mismas.

- **R_0, G_0** y **B_0** es una tripleta de números naturales, entre 0 y 255, que expresan la cantidad de color rojo, color verde y color azul, respectivamente, que existe en el color asignado al elemento de color 0-ésimo, el centro, de la ficha.

- **R_1 G_1 B_1 a R_6 G_6 B_6** son cada uno de los valores numéricos entre 0 a 255 de cada uno de los tres componentes básicos de cada uno de los colores de las 6 aristas existentes. Al menos debe escribirse uno de estos colores RGB que será aplicado a todas las aristas por igual o llegar a pintar de un determinado color cada una de ellas.

6.3.3. Sección Ficha_Var

Si usted desea simular el comportamiento de un trondentosilobite, del que sabemos que no existe y que no tiene clase, es aquí donde deberá indicar las variables extrínsecas necesarias para la simulación del mismo. Las manzanas que se desee simular, al pertenecer a la clase Manzana, no deben declarar nada por lo que dejarán esta parte en blanco.

6.3.4. Sección Ficha_Func

Si usted necesita que las variables declaradas para nuestro famoso trondentosilobite sean modificadas con cada paso de ejecución, deberá escribir aquí las ecuaciones oportunas para ello. No así, por lo dicho anteriormente, todas las manzanas o cualquier otro elemento de cualquier clase.

6.4. Cuarta parte. Secciones de parada, cambio y objetivo

Hasta aquí, el simulador puede representar modelos eminentemente matemáticos basados en ecuaciones lineales. Una curva se representa sin solución de final y la manzana cae a tierra sin mayor propósito que el ofrecido por la fuerza de la gravedad.

A partir de aquí, cualquier modelo representado se comporta buscando un objetivo. Tras este apartado se empezaron a desarrollar todas y cada una de las funciones que hacen del metasimulador un metasimulador extendido.

Antes de conocer estas funciones, conozcamos mejor las líneas de esta sección. Son las siguientes:

6.4.1. Sección Clase_Parada

Imagine que desea simular un modelo en el que caen centenares de miles de manzanas, unas en caída libre y otras lanzadas por algunos miles de manos de curiosos probadores de eventos físicos. Sabemos que, mas tarde o temprano, choque arriba o abajo, todas las manzanas caerán al suelo y ahí quedarán contemplando el espectáculo de las restantes compañeras que queden por caer o ser lanzadas.

La pregunta que surge es casi inmediata tras lo que he comentado al respecto del suelo. ¿Cuándo llegan al suelo pueden quedarse ahí sin hacer nada? La respuesta es si, mediante la escritura en este apartado de la siguiente condición:

(Manzana.Y==0)

que hace que cualquier manzana, por ser un elemento perteneciente a la clase Manzana, quede parada desde el mismo instante que satisfaga la condición.

No es la única forma de hacer que una manzana se pare. Podríamos haber escrito la siguiente ecuación en el apartado de funciones de clase:

Manzana.Y=if(Manzana.Y>0,Manzana.Y-1,0)

Que haría exactamente lo mismo, abusando de la potencia del parser en cuanto a la admisión de un operador if ternario que nos permite dar uno u otro valor a una variable en función de una condición.

La diferencia, muy importante para aquellos modelos donde se tengan unas pocas manzanas en juego pero miles de ellas sobre el suelo sin nada que hacer, estriba en que habiendo parado la manzana, no solo estará quieta sin moverse sino que el simulador la dejará aparte a la hora de realizar los cálculos de los nuevos valores de sus variables según sus funciones en cada nuevo paso. A ustedes y a mi nos es igual, pero créanme que a la computadora le viene perfecto para ahorrarse un montón de trabajo y poder representar ese tipo de modelos con agilidad.

6.4.2. Sección Clase_Cambio

Por ser una sección exclusiva para el metasimulador extendido, es aquí donde se concentran parte de las distintas propuestas que lo convierten en una aplicación bastante más rica en modelos a simular que el metasimulador convencional.

Si podemos considerar una buena mejora la condición de parada, esta es otra mejora para el comportamiento de las funciones de cada elemento. En este apartado podremos lograr que cualquier elemento de una clase de la simulación utilice la función escrita en el apartado oportuno o la función que se escriba aquí, si se cumple la condición adecuada.

Ahora cada manzana podría caer o empezar a elevarse hacia el árbol si está demasiado cerca del suelo o de otra manzana.

6.4.3. Sección Clase_Obj

En multitud de ocasiones, nuestro modelo de simulación necesitará que su comportamiento no sea tan lineal y predecible o, si lo prefiere, que sea intencionado.

Dicho de otra forma, es muy probable que cierta funcionalidad sea permitida solo si un objetivo determinado se cumple o se sigue en el camino de ser cumplido.

El objetivo perseguido es lo que se declara en este apartado. De hecho, para cerrar el círculo, aquí se declara la condición adecuada de la que hablábamos en el apartado anterior. Mejor dicho, se declara una ecuación de la siguiente forma:

$$OBJ=f(v_1..v_n)$$

Como ya se vio en las estructuras del metasimulador, cada elemento tiene una propiedad OBJ en donde se almacena la función que debe ser calculada y cotejada contra el valor de objetivo del paso anterior.

Si el valor es mayor o igual, entonces decimos que el objetivo perseguido se mejora o se mantiene y seguimos empleando las funciones de partida para este elemento.

Si el objetivo es menor en el nuevo paso que el valor del paso anterior, decimos que el objetivo no se cumple y se aplican las funciones de cambio de la clase, de forma que el elemento varía su funcionalidad hasta que vuelva a aproximarse a su objetivo.

7 FUNCIONES EXTENDIDAS

Algunos de los tipos de problemas planteados en este proyecto como modelos de ejemplo, guardan una estrecha relación con las ideas expresadas por William Flake en su libro *The computational beauty of Nature*. Este libro refleja una increíble fascinación por la naturaleza y propone distintos modelos computacionales recopilados que representan, con increíble similitud, a una buena parte de la misma.

Entre ellos hay uno que trata de una sencilla hormiga que se dedica a pintar elementos de un imaginario e inmenso tablero cuadrado o a borrarlos en función de reglas muy simples:

> *Si la hormiga tiene en frente un cuadrado blanco, se mueve a dicho cuadrado, lo pinta de negro y gira hacia su derecha.*
>
> *Si la hormiga tiene delante un cuadrado negro, se mueve a este cuadrado, lo borra dejándolo blanco y gira hacia su izquierda.*

Si observamos a esta laboriosa hormiga durante unos miles de pasos, descubriremos que ha sido capaz de pintar una sorprendente estructura de cuadrados negros y blancos. Si avisamos a una segunda hormiga para que ayude en la labor, en algunos miles de pasos mas tendremos una estructura aún más impresionante y bastante más bella de lo que cabría esperar inicialmente.

La fascinación reflejada por el autor del libro citado es compartida en casos como el mencionado, que será desarrollado en un ejemplo posterior para este proyecto. Tal fascinación ha hecho irresistible crear un modelo de abeja virtual, más adecuada que la hormiga al tratarse de hexágonos, que generase preciosas estructuras de prismas hexagonales con las que uno pudiera deleitarse.

Es probable que, como sucedió en el momento de la generación del archivo de configuración, el lector suponga que el ejemplo es tan sencillo que no hay ningún problema en describir este modelo en un metasimulador convencional.

Pero todo es ponerse a escribir dicho archivo de configuración para caer en la cuenta de que las funciones matemáticas utilizadas hasta el momento para representar parábolas y curvas en el espacio, no pueden pedir a una hormiga que pinte cuadros. Esta imposibilidad es, en parte, la responsable de este proyecto.

Si soy capaz de poder escribir modelos que admitan expresiones del tipo $Y = a + (b * Y^2)$, como expresiones del tipo $Y = Vecina(Y).Y$, entonces dispondré de un metasimulador convencional con potencia para simular hormigas, entre otros modelos de la naturaleza, o, en otras palabras, tendré un metasimulador extendido.

Pues manos a la obra. Se necesitan dos cosas tan solo: una lista de las funciones más interesantes para enriquecer hasta un nivel las simulaciones factibles y la forma de escribir las órdenes convencionales existentes hasta ahora, mezcladas con las llamadas a estas funciones de extensión.

7.1. Nomenclatura de las funciones extendidas

Este problema es típico en compiladores. Se tiene un lenguaje con una serie de palabras reservadas que nos permiten poner cosas como If, Begin, Switch y demás instrucciones del lenguaje y debemos incorporar un grupo de palabras reservadas para poder utilizar nuevos conceptos pero sin destrozar la arquitectura léxica existente.

Para poder entenderlo, nada como un ejemplo. Si en nuestro archivo de configuración tenemos una función de la forma

Manzana.Y=Manzana.Y+1

y queremos poder incluir en dicha función una llamada a la función Adyacente, que se ha creado como función extendida, podríamos tener algo así:

Manzana.Y=Manzana.Y+Adyacente(Y)+1

pero el parser no tiene ninguna función estándar ni ninguna función de usuario, que nos permite crear si las necesitamos, con ese nombre, por lo que la ejecución generará error. Además, si no es el parser el encargado de calcular Adyacente(Y) pero si debe calcular Manzana.Y+1 tenemos que ingeniárnoslas para que tanto una cuestión como la otra se resuelvan.

La solución, tras muchos ensayos y modificaciones que se adaptaran a exigencias cada vez mayores que redundaran en riqueza y flexibilidad de uso, consistió en la marcación de estas funciones mediante tokens específicos que ayudaran al metasimulador existente a saber en qué momentos debía realizar cálculos para estas funciones y cuando debía dejar al parser las expresiones convencionales.

Veamos un ejemplo que aclarará todo lo dicho. Supongamos las siguientes variables y ecuaciones de un hipotético modelo a ejecutar:

A.X=25
A.Y=27
A.Z=12
A.Vecinos=0

*A.Vecinos=**{Adyacentes(A)}***
*A.X=A.X+**{Densidad()}**-A.Y*

*(A.X==0&&**{Fichas()}**>3) A.X=1 A.Y=**{Ficha(Fpid,Y)}** A.Z=0 A.R=1*

Podemos observar que, entre las expresiones que ya conocemos, se están utilizando funciones extendidas, marcadas mediante el uso de llaves **{}**, de forma que el metasimulador en cada paso de ejecución hará lo siguiente para cada función:

- Si existe una expresión {Extended} ejecutar el cálculo de Extended como función extendida y devolver su valor ext, en una variable temporal.

- Sustituir {Extended} por el valor ext calculado, de forma que tendríamos para los ejemplos anteriores:

  A.X=25
  A.Y=27
  A.Z=12
  A.Vecinos=0

  *A.Vecinos=**1***

  *A.X=A.X+**15**-A.Y*

  *(A.X==0&&**7**>3) A.X=1 A.Y=**14** A.Z=0 A.R=1*

- Calcular el valor resultante de la expresión, tras sustitución de las funciones extendidas por los valores calculados, mediante el parser.

Una vez visto como se resuelve su uso, vamos a conocer en detalle cada una de las funciones que se han creado y lo que son capaces de calcular para que el usuario pueda utilizarlas de forma adecuada.

7.2. Lista de funciones extendidas

A la escritura de este documento, existen 29 funciones extendidas o especiales que pueden ser empleadas en los modelos de simulación.

Digo esto, porque existen unas 5 funciones mas pendientes de ser creadas, aunque podríamos tener 70 funciones escritas y otras 70 por hacer, ya que todo depende de que nos enfrentemos a nuevos modelos de simulación que requieran funcionalidad específica no contemplada o porque el usuario desee escribir funciones concretas que le simplifiquen la generación de los ficheros de configuración.

Si usted considera que debería existir una función que calcule el valor medio de una variable de clase, obtenido por el cálculo de la media de esa variable para todos los elementos existentes de esa clase, por poner un ejemplo cualquiera, bastaría con escribir esta función en el módulo independiente de funciones especiales, compilar y listos.
Debe entenderse que hablamos de un proyecto de facultad y no de un programa estrictamente comercial. Se han realizado más funciones de las necesarias pero no podía extenderse la generación de las mismas hasta el total, casi inalcanzable, de posibles funciones que un usuario quisiera tener.

Usted podrá desear disponer de una función de cálculo de la media de valores de clase como el comentado, otro usuario querrá, sin embargo, calcular la varianza, otro el camino mas corto entre fichas adyacentes y así podríamos seguir de forma que nunca estaría suficientemente completo el resultado ofrecido.

Por ello, baste saber que existe un grupo suficientemente potente de funciones que son esenciales para la gran mayoría de los modelos que puedan plantearse y que, llegados el caso de querer expandir este conjunto de funciones, su expansión sería muy sencilla e independiente del código funcional del metasimulador.

Veamos las funciones agrupadas por su funcionalidad:

7.2.1. Función Fichas()

Esta función examina el espacio de representación y contabiliza el número de prismas hexagonales que existen en ese momento, devolviéndonos un valor entero entre 0 y MaxX*MaxY*MaxZ.

7.2.2. Función Fichas(x, y, z, r)

Muy similar a la función anterior, salvo que el valor calculado corresponde solo a los prismas hexagonales que se encuentran dentro de una esfera con centro en el punto (x,y,z) y con un radio de r prismas. Digamos que nos calcula los elementos existentes en un entorno esférico local de nuestra elección.

7.2.3. Función Fichas(x, y, z - u, v, w)

De forma parecida a la función anterior, calcula los prismas hexagonales existentes en un paralelepípedo recto rectangular cuyo vértice inferior izquierdo, es decir el mas cercano al origen de coordenadas, se encuentra en (x,y,z) y cuyo vértice superior derecho se encuentra en (u,v,w) o lo que es lo mismo, los puntos extremos de la diagonal principal de este prisma rectangular.

Algunas veces querremos saber los aviones que un avión concreto ve a su alrededor y utilizaremos la esfera y otras querremos saber cuantos ladrillos se encuentran colocados en un muro, por lo que utilizaremos esta variante.

7.2.4. Función New(C, p)

Si tenemos un elemento de la clase C y queremos saber cual es la primera casilla libre que es adyacente a este prisma hexagonal, utilizaremos esta función que nos devolverá una de las tres posibles coordenadas según indiquemos en el parámetro p cual es la que deseamos que nos devuelva.

Es evidente que para los modelos de juego de vida, esta función es bastante útil, ya que nos indica las coordenadas de un hueco adyacente a un elemento dado.

7.2.5. Función New(C - i, p)

Si queremos conocer el valor de la coordenada p del hueco adyacente a un prisma hexagonal dado, pero del i-ésimo hueco adyacente en concreto, deberemos utilizar esta función.

7.2.6. Función Adyacentes(C, r)

El número total de prismas hexagonales que son adyacentes a un prisma hexagonal de la clase C dado y que lo son a un radio r de proximidad a dicho prisma hexagonal, será el valor que devuelva esta función.

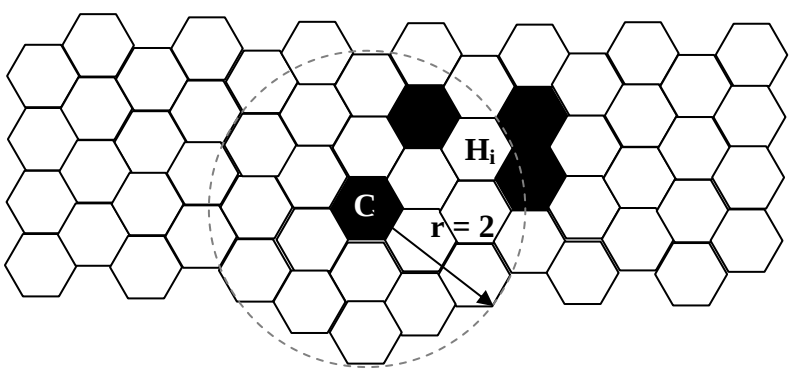
Recordemos que, en el juego de vida, un elemento no puede permanecer en la ejecución si está rodeado de un número concreto de elementos. Esta función nos ayudaría bastante para expresar la condición adecuada que, de otra forma, sería imposible de escribir.

7.2.7. Función Adyacentes(C - i, r)

Esta función es un poco engorrosa pero muy útil para un número grande de modelos.

La función nos devolverá el número de prismas hexagonales que son adyacentes al i-ésimo hueco libre adyacente a un prisma hexagonal de la clase C dado y con una proximidad de radio r.

Veamos un ejemplo en un escenario de R^2 para entenderlo mejor:



como se puede observar, la ficha de la clase C a un radio de 2 hexágonos tiene, para su i-ésimo hueco H_i, tres fichas hexagonales adyacentes a este por lo que la función nos devolverá **3** como resultado.

7.2.8. Función AdyacentesC(C, r)

Esta función es similar a Adyacentes(C, r) pero solo calcula las fichas adyacentes, en un radio r de proximidad, que sean de una clase C dada.

Cuando una persona intenta moverse necesitará saber que le rodea para no chocarse con ello, sea otra persona, una papelera o el muro de una casa. Sin embargo un depredador migrará o no de zona solo si a su alrededor existen herbívoros apetecibles, dándole igual si existen otros depredadores de su especie cercanos a él o existen plantas o rocas.

Dicho mas formalmente, en algunas simulaciones nos interesará conocer las fichas adyacentes de una clase concreta, sin preocuparnos de las adyacentes totales.

7.2.9. Función AdyacentesC(C - i, r)

Esta función es similar a Adyacentes(C - i, r) pero solo calcula las fichas adyacentes, en un radio r de proximidad, al i-ésimo hueco adyacente a la ficha de la clase C dada, que sean de dicha clase.

7.2.10. Función Densidad()

Cuando se produce un colapso en una estrella del universo, debido a un desequilibrio entre la fuerza gravitatoria y la presión de su núcleo puede suceder que, en vez de estallar como una supernova, su masa empiece a concentrarse en el espacio de una fracción de la dimensión de su núcleo, de forma que su densidad elevada pueda convertir esta estrella en un agujero negro.

Si deseáramos simular un modelo de este colapso, nos veríamos obligados a calcular en cada paso el valor de la densidad de las fichas componentes del sistema, en este caso nuestra estrella del ejemplo, frente a los huecos disponibles en el espacio de representación. Nos bastará, entonces, utilizar esta función para que nos calcule la proporción global de fichas y huecos existente en término de número de fichas por hueco dado.

7.2.11. Función Densidad(x, y, z, r)

Es posible que el cálculo de densidad no sea necesario realizarlo contra todo el espacio de simulación sino solo en una zona local del mismo.

Por ejemplo, si necesitamos conocer la atracción electrostática de una partícula puede interesarnos conocer la densidad de carga local a la misma y no al universo entero.

Si es así, utilizaremos esta función en lugar de la función general anterior.

7.2.12. Función Densidad(x, y, z, u, v, w)

Si en vez de observar la densidad de carga en el espacio local, quisiéramos calcular la densidad de carga en una sección de una pista de un circuito electrónico que se está diseñando nos interesará utilizar la función de este apartado ya que su cálculo de densidad solo tendrá en cuenta la zona del paralelepípedo recto rectangular con diagonal superior dada por los puntos (x,y,z) y (u,v,w) que son los vértices inferior izquierdo y superior derecho del paralelepípedo, respectivamente.

7.2.13. Función DensidadC(C)

En nuestro ejemplo de las cargas electrostáticas, cuando existen partículas de distintos tipos necesitaremos aislar el cálculo de la densidad de carga a un solo tipo.

De forma parecida, la habitabilidad de un entorno se dará por el número de personas que existen en el mismo, sin tener en cuenta los muebles o los ladrillos de la pared externa de la casa, por lo que utilizaremos esta función para que el cálculo de densidad sea restringido al número de fichas por hueco, pero fichas de una clase C dada.

7.2.14. Función DensidadC(C, x, y, z . r)

De forma similar a casos anteriores, en una especie de combinación de funciones de densidad, mediante esta función podremos conocer la densidad de clase, como número de fichas de la clase C por hueco, en la zona local del espacio de simulación correspondiente a una esfera con centro en el punto (x,y,z) y de radio r.

7.2.15. Función DensidadC(C, x, y, z – u, v, w)

Si el cálculo de la densidad de clase se debe realizar en el espacio dado por un paralelepípedo recto rectangular con sus vértices de la diagonal superior en los puntos (x,y,z) y (u,v,w) usaremos esta función.

7.2.16. Función Media(p)

Es bastante probable que uno de los modelos de simulación que el lector desee crear, sea el correspondiente al movimiento de los planetas del sistema solar.

Con un número pequeño de fichas de distinto color se podría simular el movimiento de los planetas en torno a nuestra estrella, el sol.

Si nos paramos a pensar detenidamente en esta simulación, debemos admitir que, estrictamente hablando, hemos reducido los planetas y su masa a un solo prisma hexagonal.

Este tipo de modelos se plantean bajo el supuesto de condiciones físicas uniformes, tanto de la materia de los planetas como del campo gravitatorio, de forma que resumimos un objeto de miles de kilómetros de radio en un solo punto, al que se le aplica una fuerza de atracción uniforme.

La simplificación hecha de un planeta como un objeto puntual no es sino la asunción de que la posición del centro del planeta basta para representarlo en condiciones uniformes. Esta simplificación requeriría el uso de esta función que no hace sino calcularnos la coordenada media de todos los elementos del espacio de simulación.

De esta forma podemos trazar una curva de movimiento del total de elementos del espacio y moverlos acorde a su centro de masas, coincidente con el centroide físico o centro geométrico, dentro de las condiciones uniformes comentadas.

La ampliación mas reciente de su funcionalidad, hace posible que el cálculo de la media pueda ser realizado sobre las coordenadas geométricas o sobre cualquier variable numérica, intrínseca o no, del elemento dado.

7.2.17. Función MediaC(C, p)

Cuando en nuestra simulación anterior de los planetas de un sistema, queramos distinguir ente planetas, satélites y estrellas, es posible que nos interese utilizar esta función que nos calcula la coordenada media pero solo de las fichas de una clase C dada.

El movimiento de las gotas de color en una lámpara que simula un volcán de lava requeriría el uso de esta función para que las curvas de movimiento de cada tipo de líquido fueran tan distintas como la lava de estas lámparas requieren.

De igual forma que con la función anterior, esta función se ha visto ampliada en su última versión, para ser capaz de calcular el valor medio de cualquier propiedad numérica existente en el elemento.

7.2.18. Función Media(C . r- p)

Si queremos ser mas precisos con nuestra lámpara de lava del ejemplo anterior, entonces deberemos utilizar esta función que calcula la coordenada media de entre las fichas de una clase C dada pero solo alrededor de la esfera de radio r que rodea a la ficha centro de esta esfera.

Como con sus análogas anteriores, la última versión realizada de esta función amplía el cálculo de la media a cualquier variable numérica de la ficha.

7.2.19. Función Media(C . r > p)

Como ya se ha comentado en apartados anteriores, el simulador tiene capacidad para leer y representar ficheros de imagen. El usuario, como en mi caso, puede querer simular modelos basados en algoritmos de tratamiento de imagen, ya sean existentes o de invención propia. Para el caso de las imágenes y posibles cálculos de valores medios se requerirá que estos cálculos se realicen únicamente en un plano y no en un espacio de representación, lugar natural de cualquier imagen convencional. Esta función será la encargada de ayudarnos en este tipo de modelos.

7.2.20. Función MaxZH (x, y, z)

Todos los que hemos ido alguna vez a una playa con nuestra bolsa y nuestra sombrilla hemos querido disponer de una función como esta.

La función nos indica el radio de la mayor esfera posible que, desde el centro en el punto (x,y,z) dado, solo contiene huecos.

En una playa, que es algo más bidimensional, nos diría en cada sitio en el que nos paráramos a calcular, cuanto espacio libre disponemos en unidades de sombrillas de radio, siempre que estas fueran prismas hexagonales.

7.2.21. Función EsHueco(C – i)

Es posible que hasta ahora no se hubiera mencionado el hecho de que el listado de funciones no sigue más orden que el de la fecha de de creación, dentro de una agrupación funcional.

Esto se comenta porque si hubieran sido ordenadas por importancia de uso en las simulaciones, esta, sería una de las primeras en cualquier lista de funciones que se pretendieran desarrollar para conferir riqueza de modelos en el proyecto.

Es pequeña y sencilla pero su importancia es vital para la gran mayoría de simulaciones.

Dada una ficha de la clase C, nos indica si la i-ésima posición adyacente a esta ficha es un hueco o no lo es.

7.2.22. Función EsHueco(x, y, z)

La importancia de la función anterior reside en el conocimiento que nos ofrece del entorno de una ficha dada. La forma en la que nos descubre este entorno tiene distintas variantes y esta es una de ellas.

Esta función nos dice si la posición de coordenadas (x,y,z) es un hueco o no lo es.

Con esta función, aún mas sencilla que la anterior, podremos conocer la naturaleza de cada posición del espacio de simulación en los términos mas interesantes para cualquier modelo de este proyecto, que, como se ha comentado, no permite las colisiones y, por tanto, impide posicionar dos elementos en el mismo hueco.

Así, si podemos saber de antemano que la ficha a mover/insertar lo hará en una posición en la que ya existe otra ficha podemos evitar la colisión. Si es un hueco, podremos disponer la ficha con éxito.

7.2.23. Función Ficha(i, P)

Seguimos con las funciones esenciales de la librería propuesta. Esta función nos devuelve la variable P del elemento con índice i en el modelo. Es bastante evidente la fuerza de esta función y su importancia se describe mejor cuando los modelos generados deben atender al orden más que a la posición entre elementos.

Como se puede suponer, esta función es la primera de dos partes. La segunda, efectivamente, nos ayudará a saber el índice de un elemento de forma que llegaremos a comprobar que, juntas, se bastan para resolver una cantidad considerable de modelos.

7.2.24. Función Indice()

Es la segunda parte a la que se hacía referencia en el apartado anterior. Como complemento de la que se acaba de comentar, esta función nos resuelve los casos en los que necesitamos saber que índice tiene asignado un elemento en la representación que se ejecuta en ese momento.

Puede decirse que esta función es complementaria de la anterior y en la mayoría de ocasiones necesitaremos hacer uso de una llamada combinada de ambas para resolver la expresión exigida por el modelo.

7.2.25. Función Indice(x, y, z)

Un poco de maniobrabilidad a la potencia de la función anterior viene dada por esta función. La función nos devuelve el índice que tiene el elemento cuya posición se sitúa en las coordenadas (x,y,z) del espacio de representación.

7.2.26. Función Entero(x)

Durante las pruebas de un modelo de simulación propuesto en el proyecto, el compilador Java comenzó a generar un error de compilación debido al uso de la función *round()* de la librería *java.math*, especializada en cálculo de valores de expresiones matemáticas comunes.

La función *round()* requiere que se le suministre una variable de tipo *single* o *real*, donde se almacena un valor perteneciente al conjunto de los números reales y devuelve un valor dentro del conjunto de los números enteros que es una aproximación al entero mas cercano en magnitud para el número real dado. En el caso comentado, se precisaba manejar la parte entera del número real introducido para poder llevar a cabo la operación necesaria en la expresión de cálculo de una de las funciones incorporadas en el modelo. Al no permitir java la compilación del programa y por tanto, no poder con el desarrollo de la aplicación por este pequeño detalle se optó por una solución que sirviera de ejemplo de la potencia de las funciones extendidas. Como se ha señalado, la potencia de las funciones extendidas es tal que permite generar funciones como las comentadas hasta ahora o, como es el caso, la que calcule el valor entero de un valor real dado.

Tal como era de esperar, se incorporó con éxito dicha función, que se mantuvo en el cálculo de la expresión del modelo comentado. Su utilidad concreta no es relevante pero evidencia la flexibilidad y versatilidad del cuerpo de funciones extendidas.

7.2.27. Función Ascendente(C, i)

Hasta el momento, la gran mayoría de las funciones intentan dar respuesta a cuestiones relativas a proximidad espacial. Si pensamos en ello, la representación gráfica existente en el metasimulador hace que sean funciones necesarias para la gran mayoría de los posibles modelos que deseáramos probar. Pero una vez resueltas la mayoría de las cuestiones que se nos puedan plantear al respecto de la proximidad espacial de los elementos, sería interesante tener alguna función que nos diera riqueza funcional en los aspectos de parentesco entre elementos.

Esta función nos indica el índice del elemento que es el i-ésimo ascendente de la ficha de la clase C dada. La ascendencia o jerarquía entre elementos se hace muy interesante en las situaciones en que queremos controlar instancias de una clase que existen por cumplimiento de alguna de las condiciones de inserción en pasos de ejecución anteriores al actual. Pensemos que cualquier problema que se desee resolver, encaminado a gestionar herencia de propiedades entre elementos, necesita disponer de esta información.

De forma parecida, si quisiéramos crear una variante del juego de vida, mediante el cual los elementos tuvieran formas distintas de fallecimiento, basadas tanto en la posición como en su antigüedad, podríamos empezar por los abuelos de los elementos mas recientes.
Como el lector supondrá, el ascendente de nivel 1 es lo que suele llamarse padre del elemento consultado, el nivel 2 corresponde al abuelo del mismo y el nivel 3 pasa al rango de bisabuelo de la ficha consultada. El ascendente 0-ésimo de un elemento es el mismo elemento, como es lógico.

7.2.28. Función Coordenada(C – i, p)

Esta función complementa parte de la funcionalidad para la localidad espacial, de forma que nos devuelve el valor de la coordenada p del i-ésimo adyacente a una ficha de la clase C dada.

Como ya sabemos, disponíamos de funciones para conocer el número de adyacentes, huecos, adyacentes a un hueco adyacente y similares, pero es esta función la que nos posibilita obtener datos sobre el elemento que cumpla la condición de adyacencia, de forma que puedan ser empleados en cálculos posteriores en el modelo representado.

7.2.29. Función Asigna(P, v)

Esta función es la encargada, como podemos deducir de su nombre, de asignar a la propiedad P de todas las fichas existentes, que dispongan de esta variable, un valor real v de nuestra elección. Esto nos permitirá ejecutar acciones del tipo $X_i=0$, (i = 0..n) en una sola orden, dentro de una única instrucción.

7.2.30. Funciones no implementadas

Es muy probable que, según haya ido leyendo las funciones existentes, el lector haya empezado a imaginar unas cuantas mas.

Por el diseño del archivo de configuración, sería muy útil disponer de una función que modificara el color existente de una ficha con tan solo indicarle el valor hexadecimal de dicho color, tal como se hace en el etiquetado html o css de las páginas Web. Ya puestos, indicarle dos colores y que realice un algoritmo de degradado de un color al otro. Y ¿por qué no obtener la traza de valores en el tiempo de alguna de las propiedades de todas las fichas o las fichas de una clase en concreto? O calcular la varianza de estas, o la desviación de esta propiedad frente a un valor o curva óptima.

Son unas cuantas que, quizás, terminen por aparecer en los modelos que se proponen al lector.

8 TIPOS DE MODELO DE EJEMPLO

Tras disponerse una versión estable del simulador convencional y a medida que las ideas relativas a una ampliación de sus capacidades tomaban forma de proyecto, comenzó a surgir el diseño preliminar de un simulador con características avanzadas frente al proyecto de partida.

Esto suponía pasar de un entorno de simulación, basado únicamente en modelos de ecuaciones lineales capaces de responder a problemas de índole puramente matemática, a un sistema de representación de modelos complejos tanto en la funcionalidad requerida como en la calidad de los modelos planteados, que debería responder a problemas sobre topología, algoritmos, sistemas naturales y un abanico tan amplio y rico como fuera posible.

El proyecto anunciado no podía quedarse en la visualización de algunas fichas siguiendo una trayectoria más o menos interesante, sino que tenía que mostrar la potencia desarrollada en la versión extendida del simulador. Se debía poder recrear los árboles de donde caían las manzanas de los primeros ejemplos, no para simular el efecto de la gravedad sino para conseguir modelar los árboles en sí mismos o representar el crecimiento de las manzanas, entre otros ejemplos imaginados y listos para ser formalizados.

Además de todo lo anterior, como proyecto, debía resolver todos los obstáculos que impedían satisfacer estos requisitos tan dispares entre sí.

En posteriores conversaciones a la de la propuesta del metasimulador y la posterior puesta en marcha de una extensión sobre el mismo, se trató el tema de los ejemplos tipo que podrían dejar nítidamente reflejada las características de esta versión avanzada de la aplicación.

Se optó por evaluar las distintas propuestas e iniciativas ideadas y bosquejadas sobre el papel e ir concretando el grupo de ejemplos que servirían al fin de mostrar la capacidad de la extensión realizada al metasimulador original.

Este preámbulo explica de la mejor manera el compromiso adoptado al respecto. Un compromiso muy simple. Quien les habla sugirió la idea, apoyada en algunos modelos de partida que servían de excusa teórica, y se comenzó la tarea de desarrollo. Posteriormente se enderezó la idea de partida para darle la consistencia esperada en todo proyecto y, principalmente, se requirieron unos cuatro o cinco modelos de ejemplo suficientemente variados que dejaran a las claras la idea inicial en la que se basaba el metasimulador extendido.

Con todo ello se comenzó a dar forma a los ejemplos de partida que pasaron la criba conjunta y los siguientes meses se compaginaron los desarrollos de funciones extendidas con la búsqueda de nuevos ejemplos o, mejor dicho, tipos de ejemplo que cumplieran lo acordado.

Los capítulos posteriores a este, que sirve de preámbulo y aclaración de los mismos, describen todos los aspectos formales de cada tipo de modelo presentado así como la narración de los distintos pormenores encontrados durante su realización.

Parece lógico pensar que si el lector se sitúa en el lugar y momento del desarrollo de cada uno de los modelos, pueda comprender mejor los distintos aspectos de cada ejemplo, tanto en la problemática particular como en la obtención de la solución concreta.

De igual manera, puede esperarse obtener una mejor comprensión de las características del metasimulador extendido, comentadas a nivel teórico, a través de la luz arrojada por la experiencia acumulada cuando se intenta resolver un caso práctico de un problema genérico.

Se debe tener en cuenta que cada modelo tipo es un pequeño proyecto en si, es decir, cada problema planteado ha requerido una investigación sobre el área a la que pertenece, un estudio de la posible solución, desconocida al inicio del modelo, distintas correcciones o mejoras de la misma y la generación de variantes de interés en cada uno de los modelos tipo.

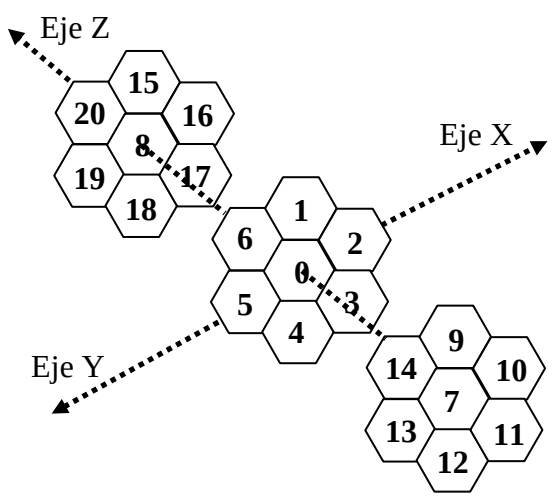
9 MODELOS TIPO I

Naturaleza
Corales, Vegetales y Esponjas

Cuando este proyecto no era más que una idea teórica, posible al menos al inicio, ya se pretendía obtener de este un modelo antes que cualquier otro. Se pretendía diseñar un helecho, un coral o, en otras palabras, una agrupación de elementos cuya estructura fuera lo mas fiel posible al esquema natural existente.

Este tipo de ejemplos parten de la idea común de un único elemento inicial que generará inserciones de nuevas fichas de la clase, siguiendo una relación de probabilidades por cada rama de inserción.

Antes que nada y como apunte teórico fundamental, veamos un prisma hexagonal y la numeración dada a sus adyacentes:



Si el elemento desde el que se realizan los cálculos es el numerado con el valor 0, entonces pueden verse los 20 adyacentes que todo elemento tiene. Así decimos que el i-ésimo elemento adyacente al dado será la ficha con valor i, tenido en cuenta el elemento marcado con el valor 0 como punto de partida.

Cambiando el elemento de partida, podría decirse que si el elemento inicial es el marcado con el número 17, entonces su sexto adyacente o su adyacente 6-ésimo es el que está marcado con el valor 8 en la representación.

> *Esta notación es utilizada a lo largo del metasimulador, sean este tipo de modelos o cualquier otro tipo, por eso es conveniente saberlo de antemano para poder utilizar las funciones en las que un índice de adyacencia es solicitado para el cálculo.*

Con estas numeraciones, está claro que una estructura de tipo arbóreo debería crecer eminentemente en las posiciones adyacentes **8, 15..20** si queremos que las ramas vayan hacia arriba. Con menor probabilidad se permitirán desplazamientos de la rama en las posiciones **1..6** y casi con ninguna probabilidad existirá extensión de la rama en las posiciones **7, 9..14** de adyacentes al elemento de partida.

Este modelo se basa en dos de las palabras del párrafo anterior. Una serie de posiciones adyacentes serán los caminos por los cuales las distintas ramas del modelo tomen forma, siempre en función de una probabilidad concreta.

Así, con un reparto de probabilidades como el dado en el modelo del coral:

Adyacentes i-ésimos P_i para caminos de las ramas del coral, con su valor porcentual de probabilidad

```
[GLOBAL_VAR]
…
P8=75
P15=75
P16=45
P17=75
P18=45
P19=75
P20=45
…
```

Probabilidad resultante para cada camino de rama P_i

```
[GLOBAL_FUNC]
P8=abs(P8-rand())
P15=abs(P15-rand())
P16=abs(P16-rand())
P17=abs(P17-rand())
P18=abs(P18-rand())
P19=abs(P19-rand())
P20=abs(P20-rand())
```

se establecen las siguientes condiciones por las que un nuevo elemento se incorpora en una de las ramas:

```
[CLASE_INS]
(A.R8==1)
A.X={New(Fpid-8,X)} A.Y={New(Fpid-8,Y)} A.Z={New(Fpid-8,Z)} …
(A.R15==1)
A.X={New(Fpid-15,X)} A.Y={New(Fpid-15,Y)} A.Z={New(Fpid-15,Z)} …
(A.R16==1)
A.X={New(Fpid-16,X)} A.Y={New(Fpid-16,Y)} A.Z={New(Fpid-16,Z)} …
```

(A.R17==1)
A.X={New(Fpid-17,X)} A.Y={New(Fpid-17,Y)} A.Z={New(Fpid-17,Z)} …
(A.R18==1)
A.X={New(Fpid-18,X)} A.Y={New(Fpid-18,Y)} A.Z={New(Fpid-18,Z)} …
(A.R19==1)
A.X={New(Fpid-19,X)} A.Y={New(Fpid-19,Y)} A.Z={New(Fpid-19,Z)} …
(A.R20==1)
A.X={New(Fpid-20,X)} A.Y={New(Fpid-20,Y)} A.Z={New(Fpid-20,Z)} …

de esta forma, cuando se hace cierta la probabilidad de que una rama del coral genere un nuevo elemento en ella, es decir, cuando $A.P_i=1$, se creará un elemento en la misma que ampliará la estructura resultante.

Este modelo crece aplicando la distribución de probabilidades de inserción en posiciones concretas desde un único elemento, es decir, tiene una idea de cómo debe ser la estructura de crecimiento pero no una ecuación fija de la misma, de forma que cada ejecución produce un elemento diferente.

También debemos tener en cuenta que los pasos de ejecución deben de ser muy numerosos para conseguir un crecimiento adecuado de la estructura ya que cada paso de cálculo no necesariamente debe arrojarnos algún resultado en la inserción.

Esto podría haber sido resuelto de forma matemática, acelerando los resultados de probabilidades mediante alguna probabilidad condicionada que nos dijera en cada paso cual de las ramas con posibilidades reales de crecimiento es la seleccionada. Pero en estos modelos se primaba, sobre los procedimientos matemáticos, la simulación de la naturaleza.

Esto unido al número elevado de elementos que puede llegar a tenerse hace de este un modelo lento de ejecución y con un elevado consumo de los recursos ya que se otorgan probabilidades a elementos que se encuentran en posiciones bajas de la estructura correspondientes a pasos de ejecución bastante anteriores.

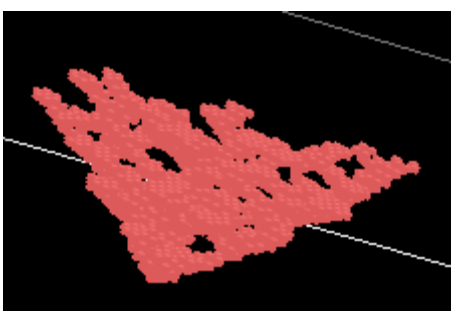
Aparte de esto, existe un detalle interesante en la modificación de la distribución de probabilidades de inserción de cada rama que puede lograr otras estructuras. Si otorgamos diferentes probabilidades observaremos como el coral puede transformarse en otro tipo de estructuras. Basta con ser algo generosos en la distribución y empezaremos a obtener unos estupendos repollos o coliflores. Si somos muy generosos, la estructura es más una hogaza de pan en el horno bajo los efectos de aumento producidos por la levadura imaginaria de esta.

Esto significa que el modelo tiene una base, una intención, de crecimiento de sus elementos exteriores, con suficiente generosidad también sus elementos interiores, que se encuentra controlado por la intensidad de la distribución de probabilidad planteada, pudiendo obtener desde relámpagos o dendritas, con escasa distribución de probabilidad casi radial, hasta las ya mencionadas hogazas, hongos de explosiones atómicas o compactas y densas esferas similares a un planeta, cuando la probabilidad es intensa y se encuentra muy distribuida entre todas las posibles trayectorias de expansión.

Nada mejor para resumir todo lo comentado que ver algunos pantallazos de distintas ejecuciones del modelo tipo comentado.

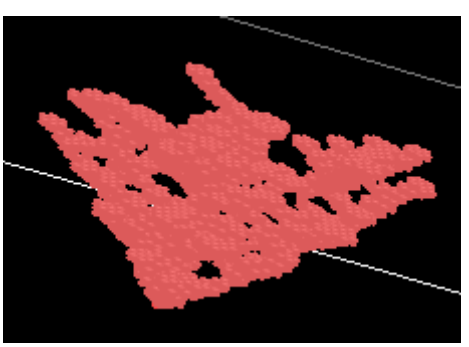
Comencemos ejecutando **Coral_1.conf**, uno de los dos ejemplos de generación de un coral, dentro de los modelos de este tipo.

Los primeros pasos de ejecución de este tipo de modelo son casi los pasos de ejecución del modelo.



Modelo Coral1 tras 57 pasos

Posteriormente la baja probabilidad y su escasa distribución hace que las inserciones se minoren de forma considerable. Así mismo las colisiones, que el simulador trata despreciándolas, suceden en mucha mayor medida que las posiciones libres para las nuevas fichas.



Modelo de Coral 1 tras 788 pasos

Si comparamos ambas imágenes, que han sido tomadas desde la misma posición, observaremos que la ganancia a partir de los pasos iniciales es muy baja y tiene un coste de computación exagerado. Casi una hora y media de tiempo se hace necesaria para ir de una a otra imagen.

Tal como se comentó, un modelo matemáticamente eficiente que estuviera orientado a la velocidad, sería capaz de producir simulaciones del modelo en cuestión de minutos pero sería poner en un microondas una carne que debe ser hecha a la brasa.

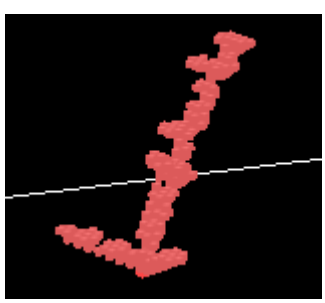
Este tipo de modelos pretende imitar en la medida de sus posibilidades los esquemas reales existentes, miles de veces más lentos que un par de horas de computación.

Probemos a variar ligeramente la distribución de probabilidades para observar como estas diferencias afectan a la generación del modelo. Para ello ejecutaremos un nuevo modelo llamado **Coral_2.conf** que presenta con respecto al ejemplo anterior las siguientes modificaciones:

Coral 1	Coral 2	Diferencia
P8=75	P8=25	-50
P15=75	P15=85	+10
P16=45	P16=46	+1
P17=75	P17=85	+10
P18=45	P18=46	+1
P19=75	P19=85	+10
P20=45	P20=46	+1

Este modelo viene a significar, respecto al anterior, un descenso considerable de las posibilidades de extensión del camino del eje superior Z y ligeros y muy ligeros incrementos de extensión por el resto de caminos existentes en ambos modelos.

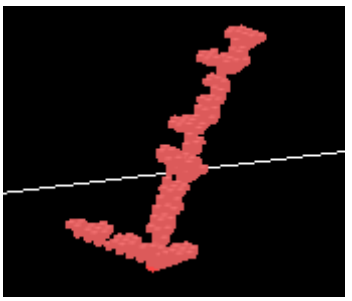
Con estas ligeras variaciones se consigue un modelo bastante diferente como puede apreciarse en la siguiente figura:



Modelo de Coral 2 tras 57 pasos

El crecimiento es extremadamente longitudinal en las tres ramas principales en las que se ha aumentado un 10% la probabilidad de expansión, una de las cuales produce una elevación muy significativa frente a las otras dos.

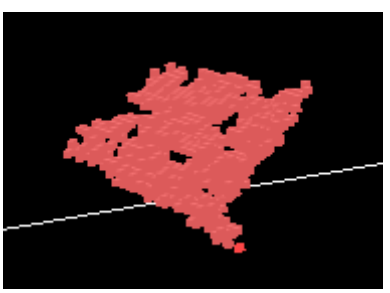
El resto de caminos apenas producen pequeñas apariciones y el eje superior Z, como sería de esperar teniendo en cuenta la bajada tan significativa de su probabilidad en comparación al resto de caminos, apenas parece existir.



Modelo de Coral 2 tras 408 pasos

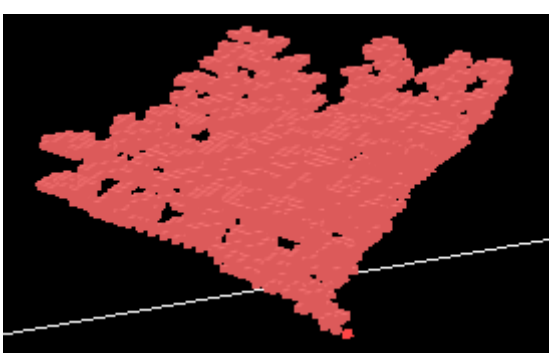
Como sucede con el primer modelo, la ejecución se resuelve en los primeros pasos sin una ganancia considerable frente al resto de pasos. Resulta curiosa esta mejora de velocidad entre modelos, pero esto es algo muy a tener en cuenta en este tipo de simulación y en cualquiera de las restantes. Es el número de fichas, apreciablemente inferior en este modelo frente al anterior, el parámetro que posibilita una velocidad muy superior, del orden de 10 pasos por cada paso en el modelo anterior.

Por último, modifiquemos la distribución de probabilidad y asignemos un 90% de posibilidades de expansión en cada uno de los caminos existentes. El modelo **Vegetal_1.conf** simula el crecimiento de lo que puede considerarse un vegetal, un arbusto, una estructura con múltiples crecimientos para el mismo número de pasos de ejecución.



Modelo de Vegetal 1 tras 57 pasos

Dadas las altas probabilidades, las inserciones crecen como se aprecia en la imagen:



Modelo de Vegetal 1 tras 93 pasos

Con lo cual se puede resumir que una variación en las probabilidades de cada camino y una elección de los caminos factibles de expansión, pueden lograr simular distintas estructuras de la naturaleza, siempre construidas con nuestro prisma hexagonal que las hace adquirir unas formas menos aparentes de lo deseado pero aún así bastante cercanas a la realidad.

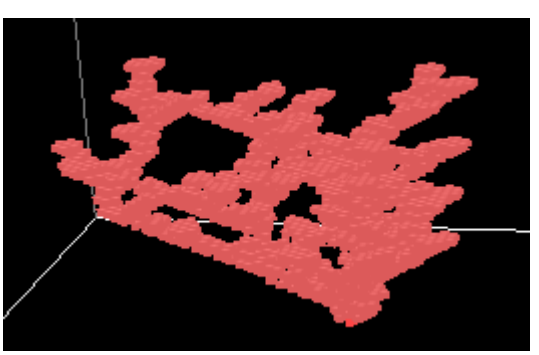
Computacionalmente, la posible mejora en los modelos para evitar que transcurra tanto tiempo hasta lograr una estructura deseada, es aquella que controla el tiempo de existencia de cada elemento de forma que aquellos prismas hexagonales con más de 50 tics de vida sean parados aunque mantenidos en la visualización. De esta forma el bucle de cálculo solo deberá tratar los elementos más jóvenes que son aquellos que procurarán una extensión en cualquiera de los caminos existentes.

Incluso esta medida es un parámetro más que perturba el modelo, ya que posibles aportaciones de nuevos elementos debido al cumplimiento de sus condiciones de inserción son ignorados por su edad.
En las distintas pruebas realizadas en **Coral_3.conf** se puede llegar a concluir que un límite de edad superior o igual a los 20 tics casi no tiene efectos apreciables en la eficiencia de la ejecución del modelo, sin embargo un límite de edad de 10 tics comienza a acelerar la simulación aunque con dos efectos.

El primero que ya hemos comentado tiene que ver con elementos jóvenes que por su edad de 11 tics, o algún otro valor superior a 10 tics, dejan de influir en la construcción del modelo de forma que no es que sea similar a alguno de los ejemplos anteriores pero mas rápidamente creado, sino que se obtiene un modelo completamente distinto a los vistos al transformar la forma de sus ramas.

El segundo efecto es que a partir de unos cientos de pasos, la coincidencia entre la escasa probabilidad y el poco número de posibles elementos de crecimiento hacen que la figura llegue a un límite en el que nada nuevo se crea y los elementos jóvenes existentes se hacen viejos y pierden su capacidad de poder insertar, como el resto de los elementos existentes.

La imagen que se muestra a continuación se obtiene en el equivalente de tiempo de apenas un centenar de pasos de ejecución del primero de los modelos:



Modelo de Coral 3 tras 54.762 pasos

Si, los datos son correctos, el modelo ha ejecutado mas de cincuenta millares de pasos en el mismo tiempo pero tiene cierta trampa.

Desde la ejecución de algún paso entre el 500 y el 750 no existían fichas jóvenes que arrojaran probabilidad como para insertar nuevos elementos. Sin nuevos prismas, los que existen o son ya viejos o pasan a serlo en no más de 10 tics con lo que el modelo muere o queda estacionario en los valores que se contemplan. A partir de ahí, sin elemento alguno que tratar, el simulador vuela paso a paso llegando a esas cifras o a centenares de veces esas, depende de lo que se tarde en parar la ejecución.

La parte más interesante de este ejemplo del tipo de modelo, es que podemos decir que no solo conseguimos recrear una estructura de la naturaleza y su crecimiento sino que, como en esta, la estructura se estaciona en una forma que podríamos llamar madura.

Si incluyéramos condiciones de eliminación, inversas a las de inserción, la estructura perecería o envejecería hasta desaparecer, casi como podría observarse en la realidad. Pero dado que quien escribe siente una fascinación considerable por los corales y planteó modelos basados en este organismo, que parece estar siempre ahí, no se incluyeron.

Eso aparte del hecho de que inicialmente era el único de los tipos de modelo que proponía quería que durase hasta que el resto de tipos estuvieran disponibles. De todas formas, se propone al lector el reto de establecer las condiciones de eliminación que irían restando elementos al modelo, una vez llegara a su madurez, hasta que este desapareciera siguiendo un orden mas o menos natural, eso si, a la velocidad de un coral.

10 MODELOS TIPO II

Hex Life Game
La Vida en un prisma hexagonal

Si sentía fascinación por un coral era, eminentemente, por el hecho de simular la vida, aunque fuera vida vegetal o casi vegetal pero muy sencilla. El hecho es sentir esa misma sensación que los programadores, los de ese grupo concreto de entre los distintos grupos en los que se puede dividir a los programadores, sienten cuando terminan un programa y siguen sus primeras ejecuciones. El programa parece mantenerse solo ante todas las adversidades y driblar cada una de las condiciones de malfuncionamiento, fallo o error que existen en su pequeño mundo.

Si, claro, los programadores somos un poco frikis. Pero imagino que esto ya es conocido por el lector.

Digo esto porque para un friki, con una simulación de coral que ve crecer, es casi una cuestión ineludible pasar a jugar con los elementos del juego de vida de Conway. Seamos sinceros, ese si es un mundo lleno de elementos simples fascinantes. Además, fijarse en la naturaleza casi te obliga a doctorarte en este juego. A intentarlo al menos.

Pues el punto de partida consiste en recordar, ahora si de forma estricta y no vaga, cuales son las reglas del juego. Hagamos un repaso del mismo:

> *El juego de vida es un autómata celular ideado por el matemático británico John Horton Conway en 1970.*
>
> *Se basa en un tablero de juego de malla cuadrada en donde dispondremos inicialmente una serie de fichas en una posición deseada. Las casillas cuadradas libres del tablero son células muertas y las casillas ocupadas por fichas serán células vivas del sistema.*
>
> *Las reglas de juego son las siguientes:*
>
> - *Una célula muerta con exactamente 3 células vecinas vivas "nace"*
> - *Una célula viva con 2 ó 3 células vecinas vivas sigue viva*
> - *Una célula viva con 0, 1, 4, 5 ó 6 células vecinas vivas, " muere"*
>
> *En cada paso de ejecución se comprueban los nacimientos, las permanencias y las muertes y cuando se conocen todas estas, se muestran en el tablero, pasándose a ejecutar el siguiente paso.*

Esta descripción corresponde al juego estándar ideado por Conway y recibe el nombre de tipo **23/3** en el que 23 significa que los nacimientos se producen si existen 2 ó 3 células vivas alrededor de una célula muerta y 3 hace referencia al número de células vivas alrededor de una célula viva necesarias para que esta última permanezca viva en el juego.

Como puede suponerse, el resto de condiciones no referidas hacen que una célula viva desaparezca en el siguiente paso de ejecución.

Desde el momento en el que uno piensa en este juego hasta instantes antes de escribirlo en un fichero de configuración, la idea que se tiene en mente refleja una especie de satisfacción por el hecho de disponer de un nuevo tipo de modelo para el proyecto. Un modelo que, además, suele apetecer a todo programador que aprecie la matemática y los juegos.

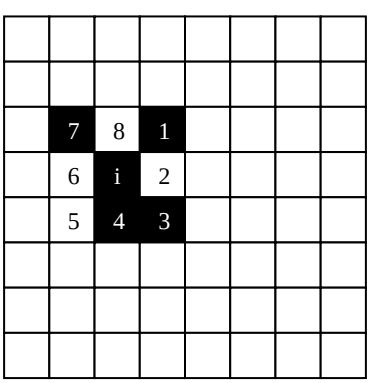
La inercia se ve acrecentada por el hecho de que, prácticamente, el modelo de coral anterior ha sido posible con un simulador convencional. Ser posible es algo que alimenta la idea de que este tipo de simulación también lo será y de una forma muy, muy sencilla. Apenas son un par de reglas básicas en un pequeño tablero.

Tras el primer intento de escritura del fichero uno llega a una sola conclusión: Esto de sencillo no tiene nada. Es mas, es imposible de llevar a cabo. Y es cierto, al menos con el metasimulador convencional.

Pero es conveniente que el lector se mantenga en esta idea de simplicidad del modelo. Si se mantiene esta hipótesis, se podrá recorrer el mismo camino de supuestos que se mantuvo en la obtención del modelo y así comprender mejor cual es el problema cuando aparezca en el planteamiento de la solución.

Tampoco parece tan difícil y, desde luego no parece imposible. Al fin y al cabo puede que sea un poco complicado definir los vecinos de una de estas células vivas, pero no parece imposible. Efectivamente, no es imposible, aunque si bastante complicado.

Si tenemos una ficha F_i, perteneciente a una supuesta clase A de células vivas del juego de vida, en un tablero cuadrado, que supondremos plano por motivos de simplificación, una posible ejecución de nuestro tablero en un paso cualquiera podría ser representada con el esquema siguiente:



de forma que sabemos a simple vista que la célula i-ésima tendrá 4 células vecinas vivas, numeradas como 1, 3, 4 y 7 y que tendrá 4 células muertas, numeradas como 2, 5, 6 y 8 respectivamente.

Esto viene a suponer que podemos recorrer las posiciones vecinas mediante las fórmulas de desplazamiento siguientes:

Posición 1: A.X+1, A.Y+1
Posición 2: A.X+1, A.Y
Posición 3: A.X+1, A.Y-1
Posición 4: A.X, A.Y-1
Posición 5: A.X-1, A.Y-1
Posición 6: A.X-1, A.Y
Posición 7: A.X-1, A.Y+1
Posición 8: A.X, A.Y+1

Y preguntarnos si las mismas tienen una ficha o un hueco en la casilla de coordenadas dadas.

Muy bien, pues nada, pregunten. Pero, ¿cómo se pregunta eso? La respuesta es sencilla. No se puede.

Si, reconozco que les he engañado, pero mas que una mentira ha sido un intento de mantenerles en la idea de la factibilidad para que chocaran de lleno exactamente con los mismos problemas con los que me encontré al intentar crear este tipo de modelo. Imagino que me disculparán por esta pequeña mentirijilla, pero creo que es una magnífica forma de ver los límites del metasimulador convencional y, por tanto, de apreciar las diferencias cualitativas existentes en este metasimulador extendido, que empezó a ganarse el nombre con esta simulación.

Es imposible porque a golpe de funciones seno, coseno, logaritmo, divisiones, sumas o exponenciaciones no se podrá resolver la cuestión a la que el metasimulador convencional no puede hacer frente pero que nuestro metasimulador extendido si debe gestionar. Veamos cual es.

Hasta el metasimulador convencional las fichas representan, con posibilidades de condicional ternario y condiciones de inserción y supresión de elementos, ecuaciones lineales que afectan a cada elemento donde se encuentren definidas. Un prisma hexagonal que se mueve según una determinada curva parabólica, no tiene conciencia del resto de elementos de su alrededor, sencillamente se mueve salvo variación de condiciones o colisión con otro elemento.

Pero ahora se necesita que la ficha, se mueva o no, mire a su alrededor, que sea consciente del entorno que la rodea, un entorno donde hay otras fichas o hay huecos donde podrá haber o no otras fichas en posteriores ejecuciones.

En resumen y para el problema que se nos plantea, **no hay forma de que el elemento i-ésimo se pregunte si existe otro elemento en la posición j-ésima respecto a su posición o si en esa posición hay o no un elemento**. No hay forma de escribir esto en el archivo de configuración del modelo a simular.

Ahora bien, si pudiéramos indicar en el modelo ciertas expresiones como las siguientes:

A.Vecinos = Vecinos(A) (= 4 en el ejemplo)
A.Huecos = 8 – A.Vecinos

tendríamos nuestra ansiada respuesta en tan solo dos sencillas funciones. La primera de ellas es la clave de todo.

Lo que se expresa en la ecuación anterior es que, mediante una función extendida, podríamos conocer fácilmente los vecinos de cualquier ficha. Por tanto, si puedo incorporar un motor de gestión y cálculo de funciones extendidas, podré escribir una funcionalidad bastante rica que, por ejemplo, logre modelizar el juego de vida para el metasimulador.

Aquí comenzó una de las ampliaciones más importantes del proyecto. Tal es su importancia frente al resto de ampliaciones que la tercera palabra que define este proyecto existe por la existencia de dicha ampliación del metasimulador convencional.

Mas que una ampliación es un nuevo mundo de posibilidades.

El siguiente paso, aparte de comenzar las modificaciones y ampliaciones pertinentes en la funcionalidad y la estructura del metasimulador convencional, consistía en mantener un equilibrio entre las funciones que pudieran necesitarse para este y otros casos concretos y que estas fueran suficientemente universales como para que sirvieran a múltiples y muy variados fines de simulación.

No se trataba de crear una función HazJuegodeVida() cuando los modelos fueran de este tipo y otra como CalculaAtractorLorentz() cuando se tratara de este tipo concreto de atractores. Se trataba de destilar en funciones la esencia de un conjunto básico, estándar y polivalente de cálculos esenciales fuera de la física o la matemática convencional y más cercana a la topología, a la algorítmica, a la idea de un conjunto de elementos interactuando en un espacio de representación.

Una vez que se cuenta con depende de qué funciones extendidas, el modelo puede ser resuelto. Veamos ahora el primero de los ejemplos, de nombre **Life_Hex_3D_1.conf**, para entender mejor su estructura.

El archivo de configuración nos muestra los siguientes aspectos interesantes:

Establecimiento de las constantes globales de nacimientos y muertes

```
[GLOBAL_VAR]
Ins=3
Del=6
```

Declaración de las variables V1 a V20 que informan del número de vecinas al hueco adyacente a una ficha dada. Declaración de las variables que informan del número de adyacentes a la ficha dada, así como si la ficha cumple las reglas de nacimiento y de muerte

```
[CLASE_VAR]
A.V1=20
……
A.V20=20

A.Ady=0
A.Delete=0
A.Insert=0
```

Funciones de cálculo del número de adyacentes a un hueco adyacente a una ficha dada. Cálculo del número de adyacentes a una ficha dada y cálculo de las condiciones de muerte y nacimiento

```
[CLASE_FUNC]
A.V1={Adyacentes(A-1,1)}
……
A.V20={Adyacentes(A-20,1)}
A.Ady={Adyacentes(A,1)}
A.Delete=(A.Ady>Del)
A.Insert=(A.V1<Ins||…...||A.V20<Ins)
```

Condiciones 1 a 20 de inserción de una nueva ficha, por nacimiento, basadas en el cumplimiento de su condición de nacimiento y que el número de adyacentes no supere el criterio de nacimiento

```
[CLASE_INS]
(A.Insert==1&&A.V1<Ins)
A.X={New(Fpid-1,X)} A.Y={New(Fpid-1,Y)} A.Z={New(Fpid-1,Z)} A.R=1
A.V1={Adyacentes(Fpid-1,1)}.....A.V20={Adyacentes(Fpid-20,1)}
A.Ady=0 A.Delete=0 A.Insert=0
.....
(A.Insert==1&&A.V20<Ins)
A.X={New(Fpid-20,X)}  A.Y={New(Fpid-20,Y)}  A.Z={New(Fpid-20,Z)}
A.R=1
A.V1={Adyacentes(Fpid-1,1)}.....A.V20={Adyacentes(Fpid-20,1)}
A.Ady=0 A.Delete=0 A.Insert=0
```

Condición de muerte basada en la variable existente para esta propiedad

```
[CLASE_SUPR]
(A.Delete==1)
```

La ejecución de este modelo realiza los siguientes pasos de cálculo:

- Calcular Adyacentes(A-i, 1) y guardar el resultado en A.Vi

- La función Adyacentes(C-p,r), como ya se comentó en el apartado correspondiente a las funciones extendidas, se posiciona en la casilla adyacente, de orden p, a la de la ficha de la clase C y realiza una comprobación de cuantas fichas adyacentes, dentro de un radio r, encuentra en dicha posición.

- Esto no es sino aprovechar el hecho de que para que en un hueco nazca una ficha, deben existir exactamente tres fichas a su alrededor. Si suponemos que la ficha dada es una de estas tres fichas, podremos observar si en su hueco de orden p, que supondremos que es el hueco que se está comprobando, el número de adyacentes del mismo es exactamente tres, en cuyo caso deberá existir un nacimiento.

- Calcular Adyacentes(A, 1) y guardar el resultado en A.Ady

- La función extendida Adyacentes(C,r) nos devuelve el número de fichas adyacentes a una ficha de la clase C en un radio r por lo que conoceremos cuantas vecinas tiene cualquier ficha existente en el espacio de representación.

- Establecer a TRUE la condición de borrado A.Delete, si se cumple que A.Ady es mayor que la variable global de muertes Del.

- Establecer a TRUE la condición de inserción A.Insert, si se cumple que alguno de los valores A.Vi es menor que la variable global de nacimientos Ins.

- Insertar una nueva ficha si, cumpliéndose la condición A.Insert, el número de vecinas al hueco i-esimo adyacente a la ficha dada es menor que la variable global de nacimientos Ins. La nueva ficha deberá calcular los valores de sus variables extrínsecas antes de ser visualizada en el modelo.

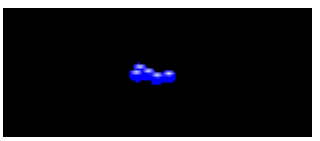
- Eliminar una ficha si se cumple la condición de muerte A.Delete

Tras la ejecución de cada uno de los puntos anteriores se dispone de un sistema que juega al juego de vida en una variante tipo 012/0123456, que significa que una ficha nace si tiene a su alrededor menos de 3 fichas vecinas y permanece siempre que no tenga a su alrededor mas de 6 fichas vecinas.

Este modelo es de alto crecimiento como puede deducirse, ya que el autómata de Conway está siendo ejecutado en el metasimulador extendido, lo que significa que las fichas no son cuadrados planos en un tablero cuadrado, sino prismas hexagonales de tres dimensiones en un tablero de rejilla hexagonal tridimensional y estas características afectan fuertemente al propio juego. Piensen que no tenemos 8 posibles vecinas en nuestro entorno, sino hasta 20 adyacentes. Piensen que no basta con examinar el plano sino un espacio de tres dimensiones.

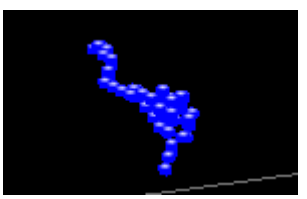
Todo ello hizo que hubiera que realizar una prueba detrás de otra hasta encontrar un modelo estable que llegara a darnos juego, de vida.

La ejecución de este modelo parte de una configuración inicial que se muestra en la imagen siguiente:



Modelo Life Hex 3D 1 en paso 0

y que, tras unas decenas de pasos, presenta una estructura estacionaria que ya no variará una vez establecida, tal como se presenta en la siguiente imagen:



Modelo Life Hex 3D 1 en paso 93

Como sabemos por el juego de vida original, este juego de 0 jugadores depende esencialmente de las fichas colocadas en el tablero inicial. Incluso para el juego original es complicado obtener estructuras que crezcan, se muevan o que oscilen entre dos configuraciones estables.

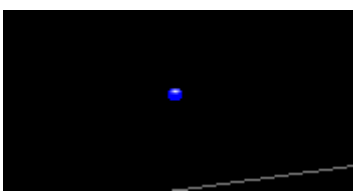
Tal es así que el propio Conway ofrecía una recompensa a quien encontrara el desplazador, nombre dado a las configuraciones que parecen moverse por el tablero sin desaparecer, de menor número de fichas.

Si es complicado en el modelo original, es extremadamente difícil dar con una configuración que sea capaz de mantener una estructura de forma continua y no digamos de forma oscilatoria. Para llegar al modelo visto se crearon decenas de ejemplos que no hacían sino perecer a los pocos pasos de la simulación, por lo cual se flexibilizaron las condiciones de nacimiento y muerte, estableciendo reglas mas acordes a la vida dentro de un mundo de prismas hexagonales.

Tal es así, que en el caso del modelo **Life_Hex_3D_2.conf**, con una configuración inicial muy similar al modelo anterior

Modelo Life Hex 3D 2 en paso 0

se llega a una solución final en la que se mantiene un solo elemento, sin posibilidades de generación de nuevos prismas, en el paso 4 de la ejecución



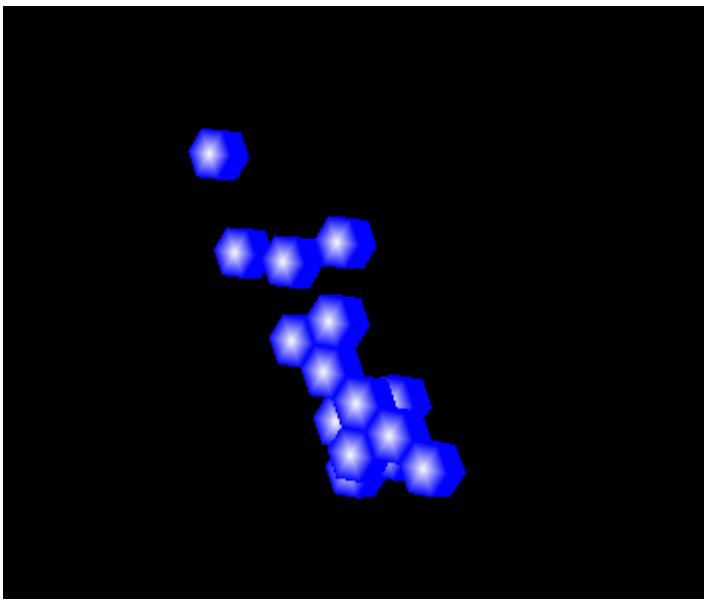
Modelo Life Hex 3D 2 en paso 4

Lo cual demuestra cuan dura es la vida en un entorno de prismas hexagonales.

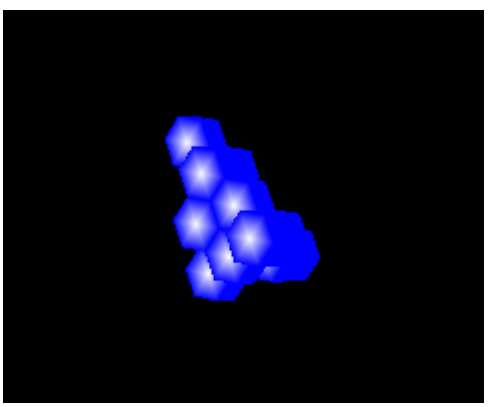
Siguiendo con el ánimo intacto, podemos probar el modelo **Life_Hex_3D_3.conf**, que intenta generar una explosión de vida superior, con la intención de que la configuración se mantenga durante centenares de pasos.

Los resultados son parecidos a los dos modelos anteriores, aunque se empieza a observar una fluctuación de elementos, entre muertes y nacimientos, prometedora.

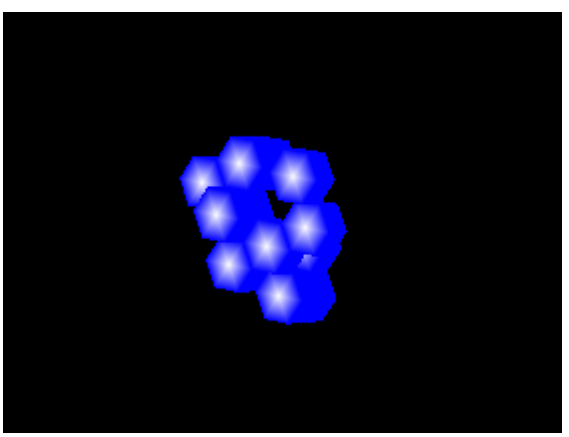
Casi se roza una estructura periódica, pero en unas decenas de pasos, la muerte diseñada en estas reglas vuelve a hacer acto de aparición, como puede observarse en las imágenes, con parte de la secuencia de ejecución del modelo, de las páginas que siguen.



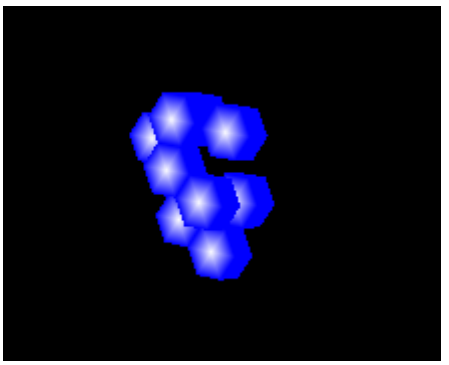
Modelo Life Hex 3D 3 en paso 1



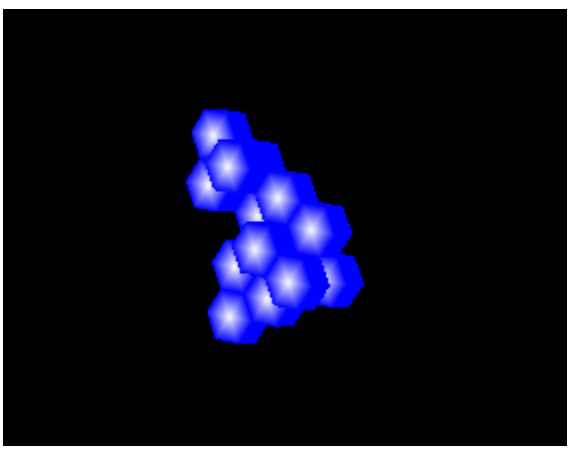
Modelo Life Hex 3D 3 en paso 2



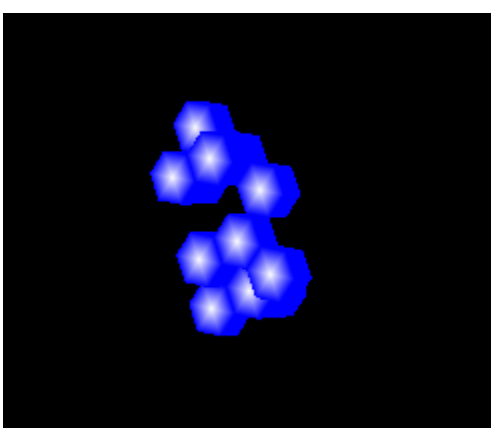
Modelo Life Hex 3D 3 en paso 4



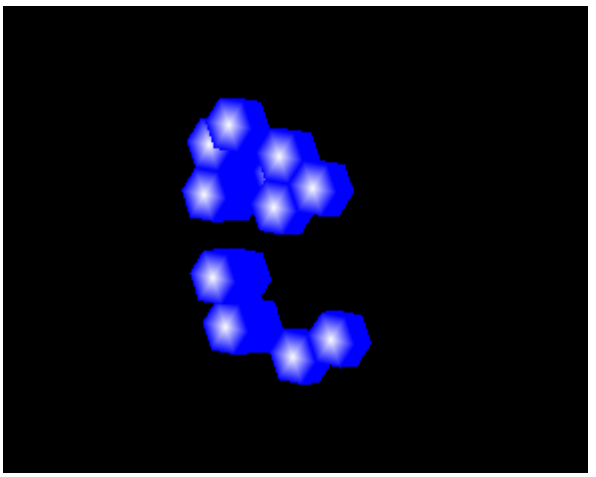
Modelo Life Hex 3D 3 en paso 5



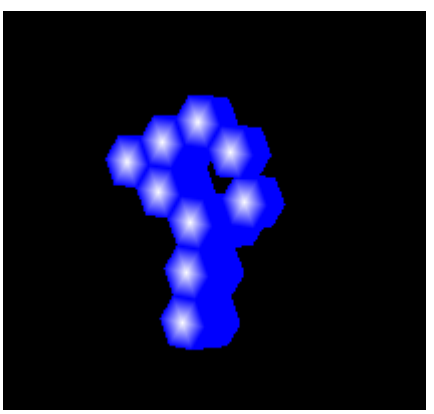
Modelo Life Hex 3D 3 en paso 6



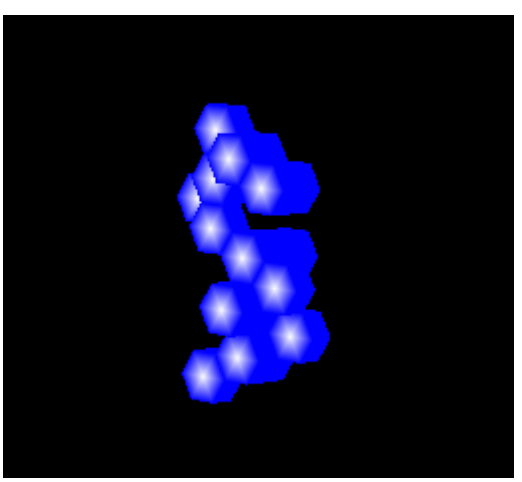
Modelo Life Hex 3D 3 en paso 7



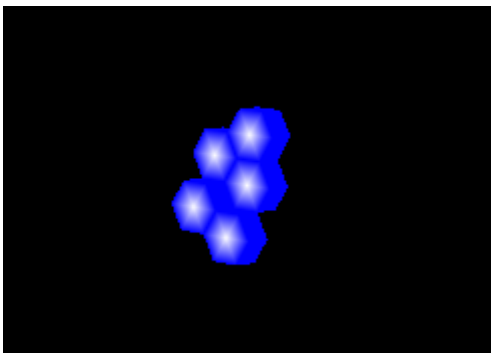
Modelo Life Hex 3D 3 en paso 11



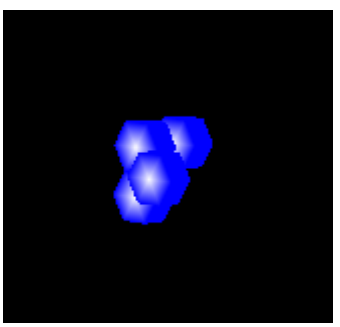
Modelo Life Hex 3D 3 en paso 13



Modelo Life Hex 3D 3 en paso 14

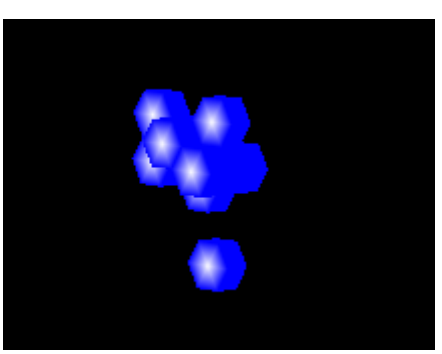


Modelo Life Hex 3D 3 en paso 22

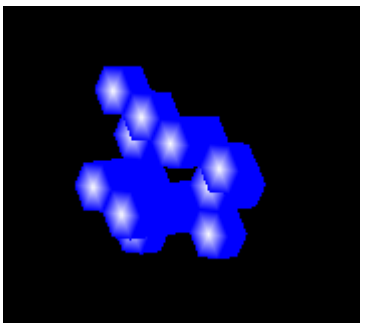


Modelo Life Hex 3D 3 en paso 27

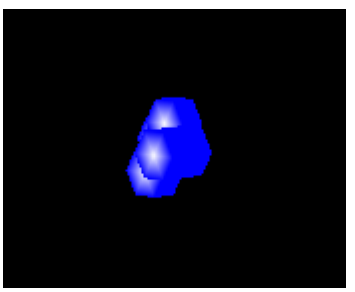
Modelo Life Hex 3D 3 en paso 29



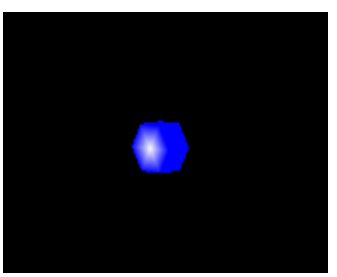
Modelo Life Hex 3D 3 en paso 31



Modelo Life Hex 3D 3 en paso 36



Modelo Life Hex 3D 3 en paso 43

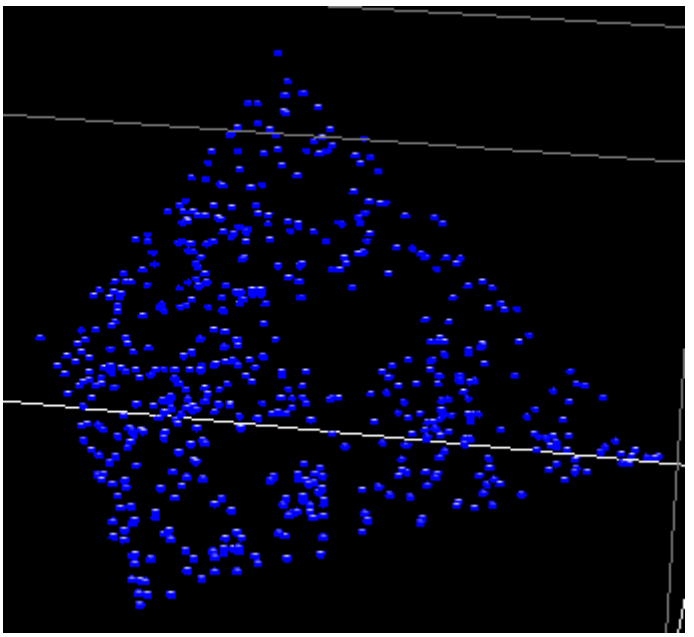


Modelo Life Hex 3D 3 en paso 54

por tanto elijamos una estructura mucho más grande en nuestro modelo **Life_Hex_3D_4.conf**, para intentar que subsista incluso con estas reglas tan duras.

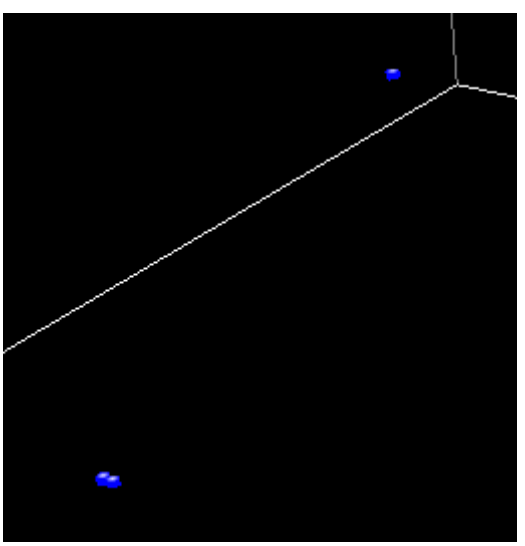
Si avanzamos hasta el apartado de los juegos de caos y volcamos la información de los elementos existentes en la pirámide de Sierpinski que se ejecuta en dicho apartado, tras dejar que esta estructura crezca unos centenares de pasos, dispondremos de un ejemplo exageradamente grande como para que las reglas de vida se mantengan entretenidas durante decenas de pasos y así poder visualizar mas tiempo un modelo de este tipo.

Tomamos, pues, el conjunto de fichas existentes en un momento de la ejecución del modelo de caos para utilizarlas como elementos de nuestro juego de vida, con la esperanza de que subsistirán decenas o centenares de elementos



Modelo Life Hex 3D 4 paso 0

y nada, la eliminación de elementos empieza a apoderarse del modelo en un solo paso, haciendo desaparecer más del 90% de los prismas en su primer cálculo de ejecución.



Modelo Life Hex 3D 4 paso 1

De estos intentos se puede deducir la relación directa existente entre las reglas del juego de vida para modelos hexagonales en tres dimensiones y el impacto en la simulación, independientemente del número de elementos y su colocación.

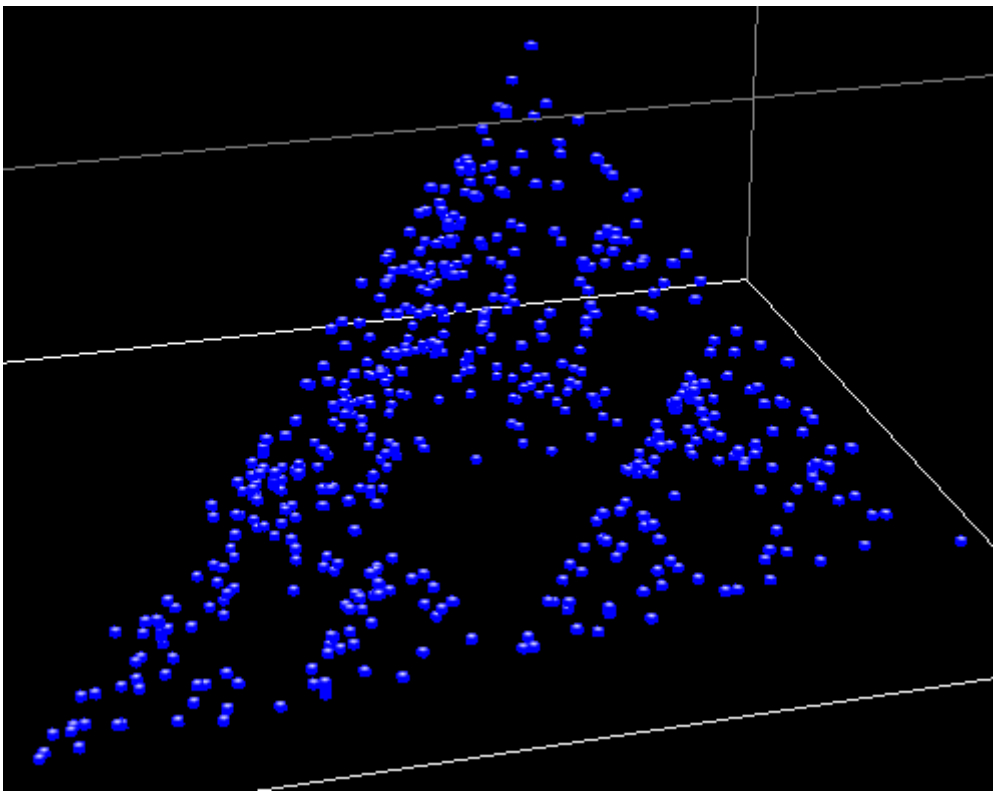
Nuestro modelo es exuberante en vida pero letal en la muerte, lo que deja un resultado neto de estancamiento en alguna pequeña isla de prismas estable en el tiempo o la desaparición del modelo hasta una situación puntual de representación.

Dentro de todas estas decenas de decepcionantes pruebas, surge un aspecto fascinante y fundamental para entender este tipo de modelos, el mismo que existe en los modelos originales de este juego. Como pequeños dioses de este universo la reflexión es clara. Si se eliminan más elementos de los que se crean, probemos a mejorar la atmósfera del sistema a ver que pasa.

En el modelo **Life_Hex_3D_5.conf**, tan solo variaremos una línea del archivo de configuración anterior, de la siguiente forma:

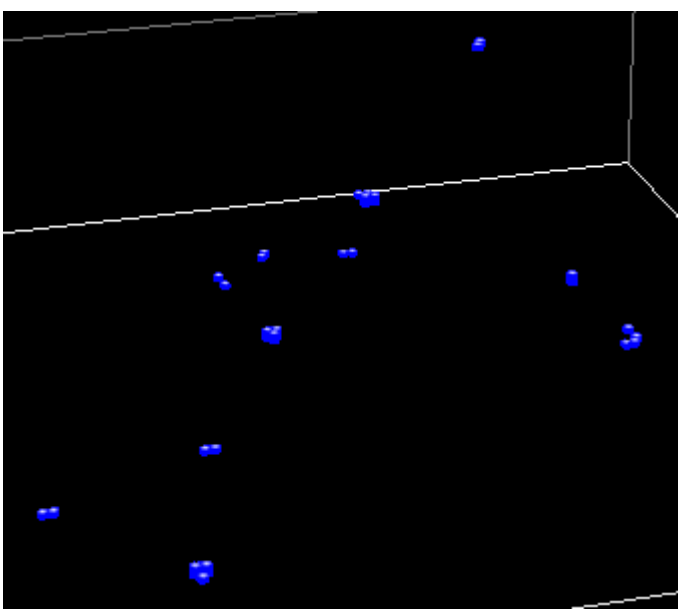
[GLOBAL_VAR]
Ii=3
If=6
*Di=~~3~~**2***
Df=6

Es decir, nuestro límite inferior de eliminación relaja su exigencia de disponer de menos de tres elementos adyacentes a tan solo 2 vecinos. El modelo inicial, evidentemente, es idéntico al anterior en su paso inicial



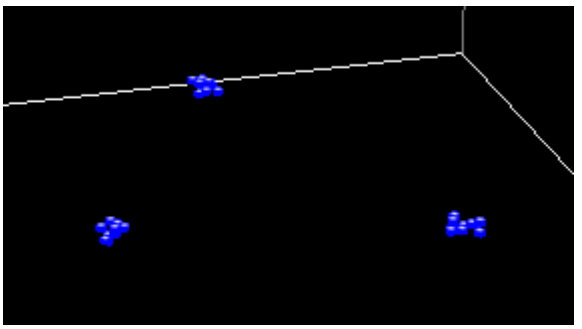
Modelo Life Hex 3D 5 en paso 0

y el siguiente paso de ejecución no es muy alentador, pues parece que las restricciones de eliminación seguirán haciendo de las suyas. Aparentemente nos encaminamos hacia otro espacio de representación sin fichas.



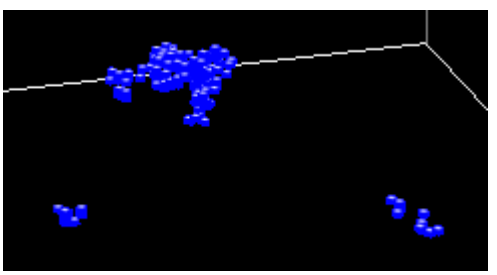
Modelo Life Hex 3D 5 en paso 1

Sin embargo, el modelo empieza a fluctuar y la relajación del límite inferior de la muerte empieza a notarse en los primeros pasos. Tal es así que la muerte reduce todos los pequeños grupos de elementos que se muestran en la imagen anterior pero no logra reducir una estructura de tres islas de elementos que parecen persistir casi de forma estable



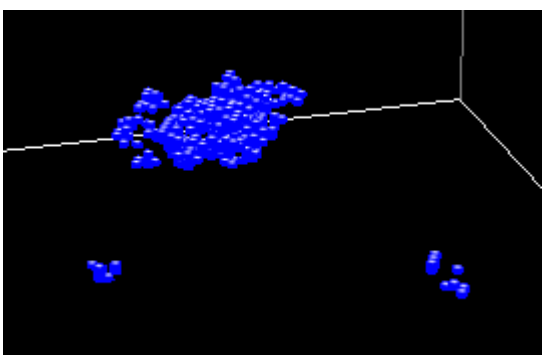
Modelo Life Hex 3D 5 en paso 15

y no solo no logra reducir el triplete de vida, sino que estas islas empiezan a burbujear de forma oscilatoria inicialmente para empezar a crecer alrededor del paso 100 de ejecución del modelo. La isla central empieza a tomar forma de estructura candidata al premio de la permanencia en la ejecución, a juzgar por el número de pasos recorrido



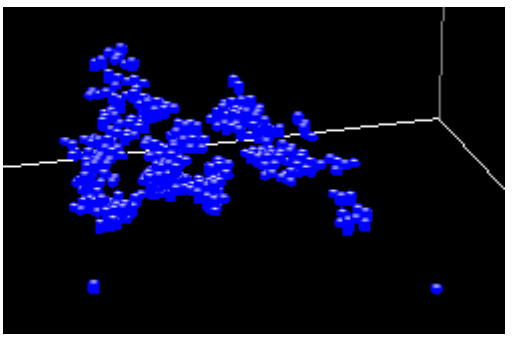
Modelo Life Hex 3D 5 en paso 138

y cuando la ejecución lleva recorridos casi doscientos pasos parece que el modelo tomará una única estructura basada en una isla central creciente y dos pequeños satélites tal como puede observarse en la siguiente imagen



Modelo Life Hex 3D 5 en paso 192

Si doblamos el número de pasos podremos comprobar que el modelo comienza a crecer en su zona central, de forma que termina por abarcar las dos estructuras vecinas y crece generando fracciones de esta isla central que parecen volver a la configuración de tres segmentos o incluso de cinco segmentos de parecidas dimensiones a las de partida, quizás con mayor extensión



Modelo Life Hex 3D 5 en paso 464

y a un ritmo cada vez mas elevado, con lo que todo apunta a que la vida, como dicen en esa película de reptiles prehistóricos, se abre paso tras nuestra pequeña generosidad en las reglas. Las mismas reglas que aguardan a ser variadas con tiempo y paciencia para obtener distintos modelos, tan interesantes como los vistos en este apartado.

11 MODELOS TIPO III

VBees
Las abejas virtuales

Los dos primeros tipos de modelo comentados se alinean en una de las dos partes de una raya que ya fue comentada en capítulos anteriores de este documento. Me encontraba generando modelos que mejoraran la colección de ejemplos para cada tipo cuando tocó una de nuestras habituales reuniones de los viernes con nuestro tutor.

En ella se me insistió, afortunadamente, una vez mas en que me quedara unos días con un libro sobre la belleza computacional de la naturaleza. Sinceramente, no era por hacerle ascos al libro sino que mi forma de ser me obliga, tozudamente en ocasiones, a buscar ejemplos y modelos por mi cuenta sin buscar ayuda en ninguna fuente. Me parecía más deportivo que fisgar en un libro.

Digo afortunadamente porque en una sola semana de lectura de sus hojas, encontré algún material que parecía bastante interesante como aplicación para el metasimulador extendido. Había algunos candidatos a tipo de modelo bastante interesantes. Uno de ellos, que me llamó considerablemente la atención, trataba sobre el recorrido de una hormiga, con un cubo de pintura negra, una brocha y un borrador, a través de un espacio de malla cuadrada, en teoría infinito.

Tal como ya he adelantado en otros capítulos, si el libro tenía una hormiga pintora en una malla cuadrada plana y mi malla era hexagonal espacial, la lógica mandaba que mi pintora fuera una abeja. Es evidente, ¿no creen?

De nuevo, el primer reto era adaptar un modelo de autómata basado en movimientos ortogonales en el espacio de R^2 a otro modelo basado en movimientos sexagesimales en el espacio de R^3 con algo de experiencia al respecto, gracias al tipo de modelo del juego de vida que vimos en el apartado anterior.

De nuevo el atrevimiento, supongo, hacía que el esquema de resolución mental que diseñaba mientras daba vueltas por la habitación fuera factible y, es más, sin mayor inconveniente que crear el archivo de configuración adecuado. Listo en unas pocas horas y a otra cosa. De nuevo otro error de cálculo. Veamos porqué.

El algoritmo utilizado por una hormiga virtual decidida a pintarnos un infinito tablero cuadrado de casillas blancas es el siguiente:

> *La hormiga mira hacia la casilla que tiene en frente.*
>
> *Si está en blanco, se sitúa en dicha casilla, la pinta de negro y se gira para mirar la casilla que se encuentre a la derecha de esta.*
>
> *Si está en negro, se sitúa en dicha casilla, borra la casilla y se gira para mirar la casilla que se encuentre a la izquierda de esta.*

Es un algoritmo extremadamente sencillo para lo que es capaz de conseguir. Si ponemos a nuestra hormiga a ejecutar los primeros millares de pasos, se empieza a vislumbrar una estructura de trazado que solo puede dejarte con la boca abierta por lo sorprendente de su arquitectura. Si realizamos este mismo algoritmo con dos hormigas, el resultado deja pequeño al anterior, ya de por si espectacular, y empieza a asemejarse a los mejores trazados árabes de los techos de la Alhambra granadina.

Es posible que, como en mi caso, el lector imagine que no existirán problemas en crear un archivo de configuración para este tipo de modelo. Si ya contamos con funciones extendidas para los aspectos mas relevantes de la adyacencia entre elementos y la gestión de huecos en el espacio de simulación, parece sencillo indicar a nuestra abeja virtual que mire si lo que hay a su derecha tiene color, un prisma hexagonal, o no, un hueco, mirar esto mismo pero en la posición contigua del recorrido y, no digamos, mover nuestra abeja de una posición a otra.

Todos los inconvenientes descritos pueden ser resueltos por las funciones matemáticas convencionales y parte de las funciones extendidas ya desarrolladas, pero hay una cuestión que nos impide ejecutar este tipo de modelos. Encima la cuestión es tan sencilla, que pasa desapercibida al pensamiento, hasta que uno intenta diseñar el modelo.

Por lo que sabemos del algoritmo tenemos una abeja pintora y mira y, en función de lo que encuentre, pinta o borra casillas de un tablero espacial. Ahora piensen en la abeja encima de una casilla, ya sea este blanco o negro. Ese es el problema, no puede haber dos elementos en la misma posición y hay dos elementos en una misma posición. Porque la cuestión esencial para la resolución estriba en que la abeja es un elemento de la simulación y la casilla también lo es. Ya sabemos que esto es una colisión en nuestro metasimulador. Desde luego no es una simulación de nuestra abeja virtual.

Ya son bastantes ocasiones en las que se supone un desarrollo sencillo y se topa con los pequeños detalles. Son tantas que puede decirse que esos pequeños detalles no son tales. De hecho no es un problema sencillo sino una cuestión trascendental que guarda relación con la estructura de diseño básica del proyecto. Si recuerdan, se intentaba llegar tan lejos como se pudiera.

Si avanzar mas supone expandir el conjunto de funciones extendidas, siempre que no sean funciones específicas para un problema, se desarrollan y se sigue avanzando. Pero la materia empieza a reclamar aquello de lo que hablábamos en los primeros pasos de este proyecto. No, hasta aquí se puede llegar pero no se puede llegar mucho más por este camino. Existen grupos de algoritmos que necesitan autómatas con una configuración imposible de simular con las bases existentes.

No, este simulador no podría jugar al ajedrez y no porque no pueda esperar una reacción de un segundo jugador, necesario para este tipo de algoritmos, sino que no puede ni es cometido del mismo afrontar la resolución de árboles de juego y su poda heurística para gestionar las actualizaciones de información de una base de hechos, necesarios en juegos como el indicado.

Esto ni es una claudicación ni una observación negativa de este metasimulador, sino el marcaje de uno de los límites de la resolución, que sigue siendo suficientemente inmensa como para no preocuparse de este tope. Esto pretende ser una explicación de la naturaleza de los, aparentemente sencillos, obstáculos a los que hacía referencia.

Volvamos a nuestro tipo de modelo, porque nadie ha dicho que no sea resoluble. De hecho, con una pequeña ocurrencia bastó para dar vía libre a estas simulaciones. La solución es sencilla como sencilla parecía la barrera que nos impedía simular este sistema. Pongamos una sombra de una abeja, unos niveles del plano Z mas abajo y dejemos los focos a las casillas, rellenas o no. Si la abeja revolotea en Z = 2 y las casillas se iluminan o no en Z = 5 no sucederá nunca la tan temida colisión por disponer ambos elementos en una misma posición del espacio.

Dicho y hecho, ya podemos presentar el modelo **Bee.conf**, que ejecuta el primero de nuestros ejemplos. Su archivo de configuración, que examinaremos en detalle, describe lo comentado anteriormente:

> *Variables de Clase. La Clase representa tanto a la abeja como a las casillas que pueden ser pintadas o borradas. La variable id nos sirve para identificar a esta abeja con id=0 lo que se representa como A.id=0. La variable Insert se utilizará solo para asignar los valores 1 ó 0 a la abeja, siendo 0 en todos los demás. La variable Did nos dirá el id de la casilla que deba ser eliminada. La variable Dir indicará el sentido, a la derecha o a la izquierda, que tome nuestra abeja virtual.*
>
> *[CLASE_VAR]*
>
> *A.id=0*
> *A.Insert=0*
> *A.Did=-1*
> *A.Dir=1*
>
> *Funciones de Clase. Cada elemento almacenará en la propiedad extrínseca id el valor de su identificador como objeto Hex, propiedad llamada Id, de forma que se sabrá quién es la abeja, Id=0=A.id y quienes son casillas, A.id=Id>0 de forma que las siguientes funciones exclusivas para la abeja, le sean aplicadas solo a esta.*
>
> *El cálculo de Insert se hará en función de si la proyección de la abeja se encuentra delante de un hueco, valor 1, o no, valor 0, siendo 0 para el resto de elementos, ya que son casillas. Nos referimos a proyección de la abeja y no a la abeja en sí porque lo que se calcula es respecto a la posición de la abeja pero un plano de altura mas arriba.*
>
> *Es decir, la abeja se encuentra en Z=z y nosotros observamos la casilla (x,y,z+1) para evitar disponer la abeja en colisión con alguna casilla. El cálculo de la variable Did se realiza, para la proyección de la abeja en caso de que no exista inserción y, por exclusión, exista eliminación de una casilla, de la casilla que se encuentra frente al vector de recorrido de la abeja pero en el plano Z=z+1 de las casillas.*

Con el valor Dir existente se calculan las nuevas coordenadas tridimensionales (X,Y,Z) de la abeja. El valor de Dir es actualizado para mover la abeja, su vector de visión, a la derecha, si hay inserción de una nueva casilla, o a la izquierda, si por el contrario se produce la eliminación de una casilla existente.

```
[CLASE_FUNC]

A.id={Indice()}

A.Insert=if(A.id==0,if({EsHueco(A-{A.Dir+14})}==1,1,0),-1)

A.Did=
        if(
                A.id==0&&A.Insert==0,
                        {Indice
                                (
                                        {Coordenada(A-{A.Dir+14},X)},
                                        {Coordenada(A-{A.Dir+14},Y)},
                                        6
                                )
                        },
                        -1
        )

A.X=if(A.id==0,{Coordenada(A-{A.Dir},X)},A.X)
A.Y=if(A.id==0,{Coordenada(A-{A.Dir},Y)},A.Y)
A.Z=if(A.id==0,5,6)

A.Dir=
        if(
                A.id==0,
                        if(
                                A.Insert==1,
                                        if(
                                                A.Dir==6,
                                                        1,
                                                        A.Dir+1
                                        ),
                                        if(
                                                A.Dir==1,
                                                        6,
                                                        A.Dir-1
                                        )
                        ),
                        -1
        )
```

Condición de inserción. Si la abeja, único elemento que puede obtener un valor 1 en la variable Insert, ordena una inserción, entonces se crea una nueva casilla en la posición de la proyección de la abeja, que se encuentra en el plano Z=z+1, que es el plano donde se disponen las casillas.

```
[CLASE_INS]

(A.Insert==1)

    A.X=Fpid.X

    A.Y=Fpid.Y

    A.Z=6

    A.R=1

    A.C0r=rand(Fpid.C0r)+5

    A.C0g=rand(Fpid.C0g)+5

    A.C0b=rand(Fpid.C0b)+5
    .....
    A.C6r=Fpid.C6r+5 A.C6g=Fpid.C6g+5 A.C6b=Fpid.C6b+5
```

Condición de borrado. Si la abeja, elemento con id=0, tiene un valor para su variable Did de i, entonces se elimina el elemento con id=i, que corresponderá siempre a una casilla existente.

```
[CLASE_SUPR]

(A.id=={Ficha(0,Did)})
```

Posición inicial de la abeja en el plano Z=z, un nivel inferior al plano Z=z+1 donde se insertan las casillas.

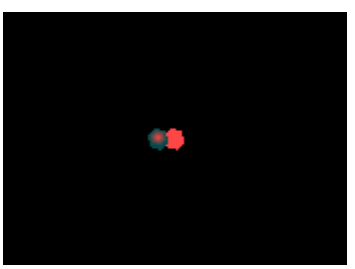
En nuestro ejemplo, la abeja se mueve en Z=5 y las casillas se insertan en Z=5+1=6 para evitar colisiones entre estos elementos.

```
[FICHA_DEF]
0 50 50 5 1 A

[FICHA_COLOR]
0 1 2 255 70 70 255 70 70
```

De esta forma nos encontramos con un modelo inicial en el que solo existe nuestra abeja, como se muestra en la siguiente imagen:

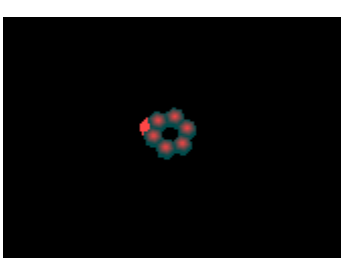
Modelo Bee en paso 0

Al inicio de la ejecución, nuestra abeja comienza a mirar el nivel superior del tablero para observar, como si estuviera en dicho plano, si ve casillas vacías o pintadas. Dado que no existen casillas, pinta la primera y se mueve a su derecha.



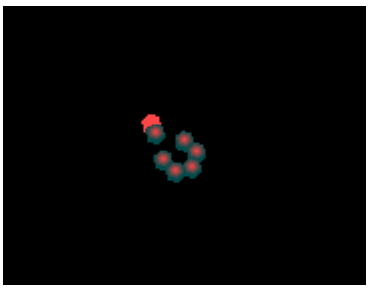
Modelo Bee en paso 2

Tras ello, sigue viendo casillas vacías por lo que sigue pintando estas y girando siempre a su derecha, hasta que llega a la primera de las casillas coloreadas.



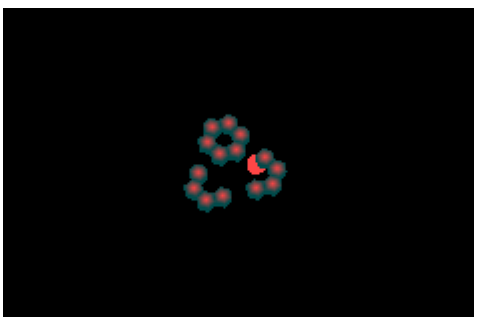
Modelo Bee en paso 7

Dado que tiene delante una casilla pintada, la primera casilla que pintó nuestra abeja, decide borrarla pero ahora gira a su izquierda, pintando en esa posición una nueva casilla y girando de nuevo a la derecha.



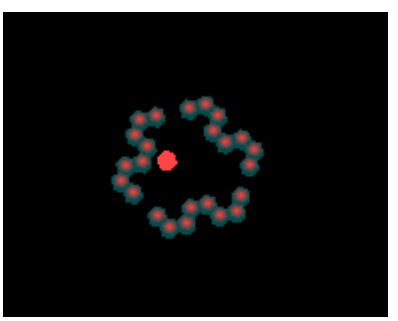
Modelo Bee en paso 9

Con estas simples reglas de pintado o borrado y su consecuente corrección de trayectoria, empezamos a ver como la abeja diseña un curioso tapiz de casillas de color.



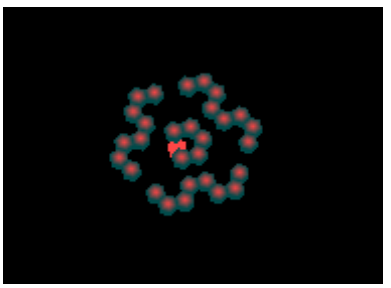
Modelo Bee en paso 27

Parece que nuestra abeja costea el perímetro dado por las casillas de color de forma que en torno a este, realiza el siguiente anillo de casillas coloreadas.



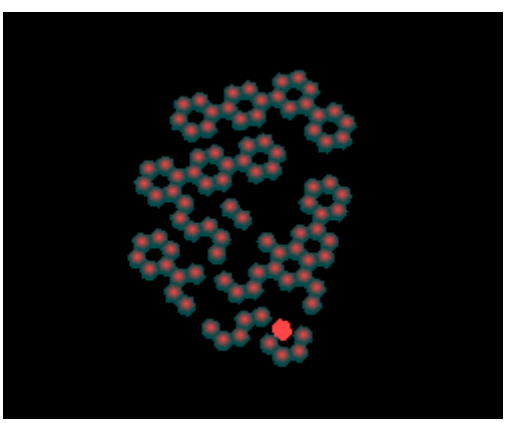
Modelo Bee en paso 73

Pero vuelve a situarse en las posiciones centrales antes de expandirse hacia fuera de nuevo.



Modelo Bee en paso 78

Si damos rienda suelta a los pasos de ejecución de nuestra abeja comenzaremos a tener una estructura mucho más rica en formas.



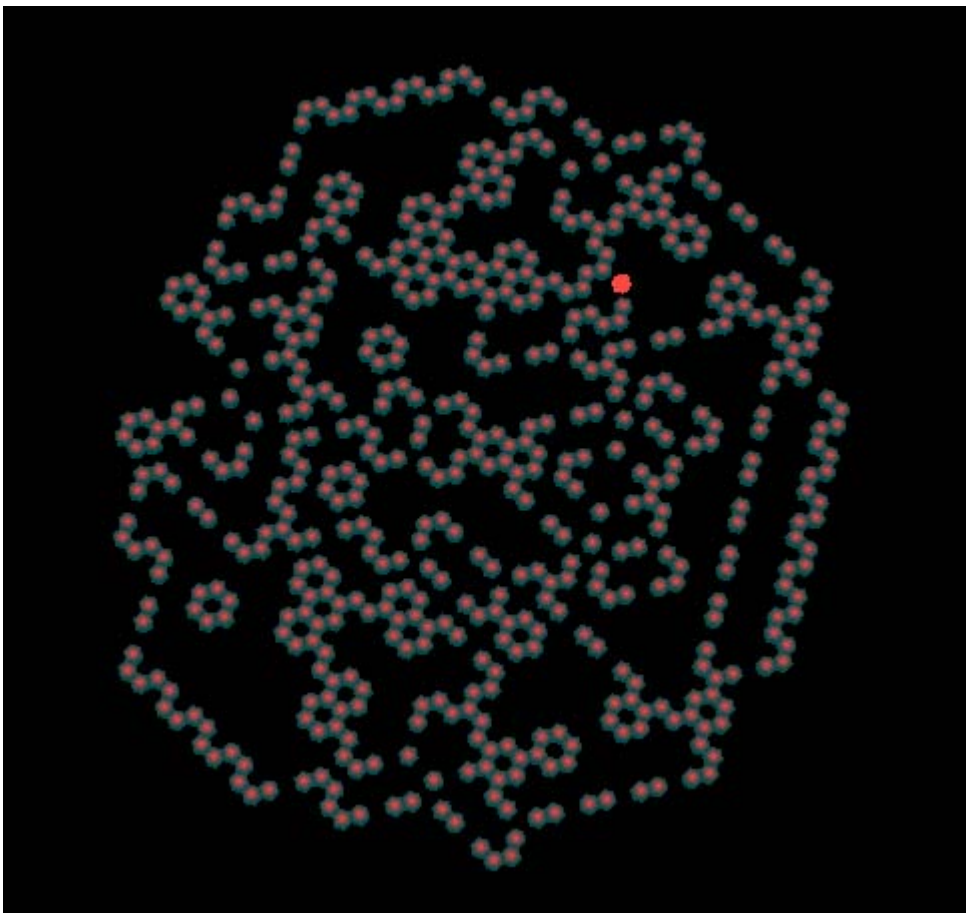
Modelo Bee en paso 1.123

A partir de aquí, dejamos que el metasimulador represente unos cuantos miles de pasos para que la abeja pueda ir borrando y creando distintos elementos, de forma que pueda ser observada la estructura que empieza a surgir con motivo de las fases de ensanchamiento del contorno de elementos y su correspondiente ribeteado en modelos hexagonales que trazan cada una de las aristas de los distintos anillos de trazado.

Este comportamiento va generando nuevas fronteras en el modelo, que son revisadas en una especie de doble paso, uno que delimita la nueva expansión y otro que comiena a formatear este contorno.

Como es lógico, cada paso de ejecución ralentiza el resultado final, dado que el recorrido se amplía de forma lineal pero contínua. Nuestra abeja comienza a representarnos el incio de lo que parece una figura en expansión sin forma aparente.

Con un número de pasos adecuado, vemos como el simulador llega a modelos geométricos en los que queda acentuada una cierta autosemejanza constructiva y una estructura lógicamente muy hexagonal, tal como el espacio de simulación y las reglas de movimiento parecen requerir.



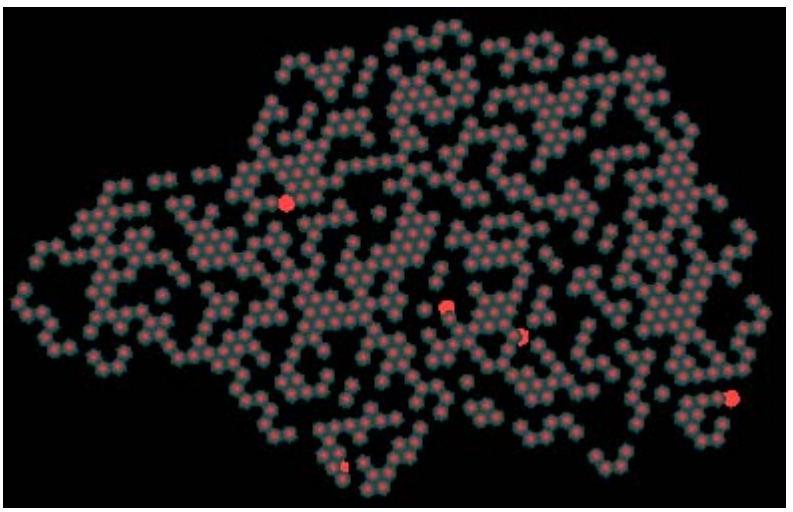
Modelo Bee en paso 18.681

Una vez este tipo de modelo se encuentra en perfecto funcionamiento, tal como se ha comprobado en los párrafos anteriores, se puede empezar a dejar volar la curiosidad y generar modelos mas complejos que los vistos hasta ahora.

Esta complejidad parece evidente con tan solo proponer simulaciones en las que mas de una abeja realizé las tareas de pintado.

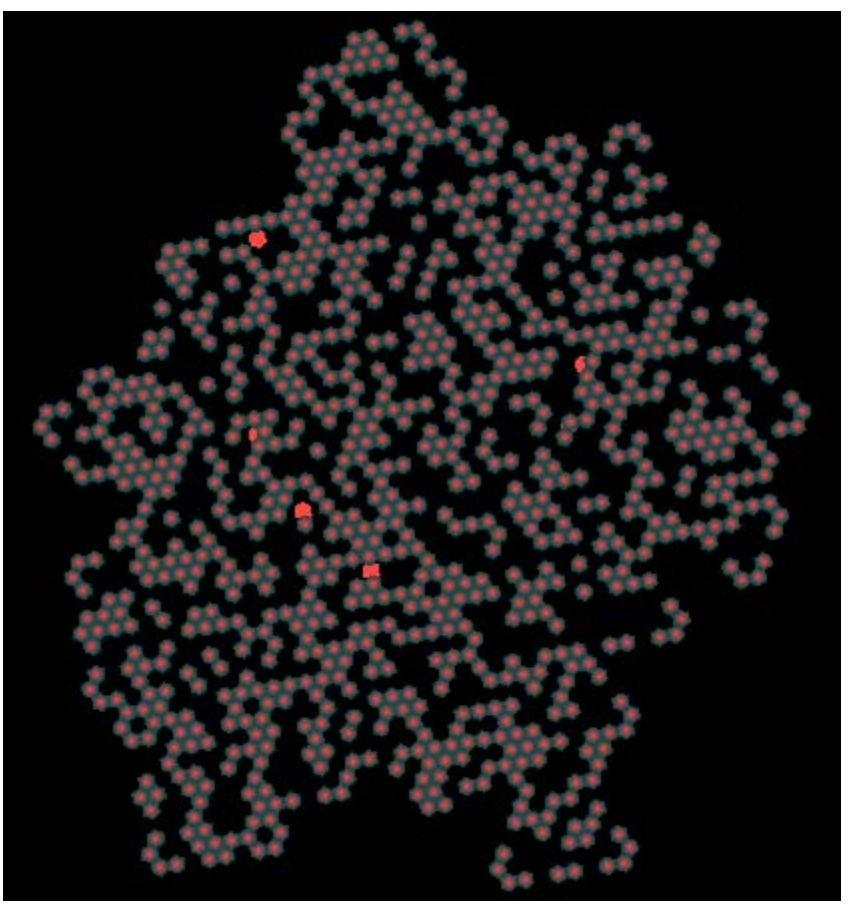
Si se piensa, nuestra mayor dificultad ahora consistiría en la posible colisión entre compañeras que se mueven en el mismo plano, pero dado que nuestro simulador gestiona las colisiones, aunque no las permita, podemos saciar nuestra curiosidad por conocer otras posibles formas geométricas resultantes del empleo de mas pintores en nuestro espacio de representación.

Con cuatro abejas más, modelo **Bees_1.conf**, las diferencias son notables



Modelo Bees_1 en paso 962

Si variamos sus posiciones iniciales, modelo **Bees_2.conf**, la geometría cambia



Modelo Bees_2 en paso 1.659

A partir de aquí, es el lector quien puede probar aquellas combinaciones de pintoras y posiciones de las mismas, inclusive de variación de sus vectores de recorrido, que puedan ofrecer multitud de distintos patrones hexagonales para su contemplación y disfrute. Tan solo debemos mantener a nuestras abejas, como Miguel Ángel en la Capilla Sixtina, pintando casillas por encima de sus cabezas.

12 MODELOS TIPO IV

Particle Swarm Optimization
Sincronía entre pájaros

Siempre se han realizado los anuncios televisivos con la intención de provocar el interés del telespectador por el producto anunciado y, si es posible, el deseo compulsivo por su adquisición aunque quien les habla no se haya distinguido por pertenecer a dicho grupo, lo que no quita para compartir el interés, al menos en unos pocos anuncios. Tan interesante como el famoso anuncio de BMW, en el que aparecía una simple mano moviéndose al viento mientras preguntaban si nos gustaba conducir, me resultó otro anuncio, también de un automóvil, al respecto de la capacidad de reacción de una bandada de pájaros en movimiento. Sin llegar al deseo de poseerlo, reconozco en el segundo que me animó a adquirir, una forma distinta de compra, conocimiento sobre este tema en el cual ya había pensado, en algún que otro rato perdido, años atrás.

El famoso libro que ya les comenté en apartados anteriores, en el que se comentaba este fenómeno, llegó un poco tarde y solo pudo ayudarme a corroborar lo recopilado en Internet. Para cuando el libro me habló de los pájaros, ya me encontraba dándole vueltas al tipo de modelo que nos ocupa en este capítulo.

Es evidente que si las pequeñas formas del juego de vida de Conway me llamaban la atención o si tiendo a quedarme absorto viendo la línea directriz del caparazón de un Nautilus o las formas de las estalactitas, para los que no encontré un tipo de modelo atractivo que incluir en este proyecto todo sea dicho, es muy probable que me embobe con igual intensidad con una bandada de pájaros.

No tiene sentido ese comportamiento, como no lo tiene la localización de objetivos que hacen las hormigas por el suelo o las abejas en el aire. Al fin y al cabo no son inteligentes como nosotros y ustedes y Yo sabemos que la salida de un estadio ya muestra de por si hasta donde alcanza nuestra inteligencia. Si, es cierto, nosotros somos muy individualistas y estos animales trabajan bastante bien en equipo, pero aún así es curioso que naveguen por el aire y a la velocidad a la que vuelan, mientras nosotros convertimos en una odisea de ineptitud que unas decenas de personas crucen ambos extremos de una calle cuando el semáforo de la calle nos lo permite.

Es probable que, como sucede con los tiburones y su campo electrostático de adquisición de objetivos, dispongan de poderosos atributos naturales que les permitan realizar esas maniobras, pero no, nada de eso. Un pájaro como el que nos imaginamos solo tiene dos ojos, alas y una pequeña cabeza para organizar todo esto.

Parece ser que el truco de esta maravilla reside en la organización, no de un pájaro, sino del total de pájaros que forman el grupo. Parece ser que su organización les relega a meras partículas dentro de esta nube de compañeros, cuestión que no quita para que siga siendo enormemente atractivo verlos y entender cómo lo consiguen.

No, tampoco llevan comunicadores, de forma que puedan disponerse acorde a los edictos de control central, una especie de Radio Golondrina FM, o por debate entre sus miembros, una especie de Congreso de los Abejarucos, ya que no hay tiempo para todo eso en el rango de las décimas de segundo.

La forma más sencilla de explicar la base de su comportamiento, utilizando algún ejemplo del nuestro, sería pensar en usted y Yo en el mismo coche, volviendo de la facultad a algún punto común de Aluche. Cuando abandonamos el recinto, nos incorporamos a una vía de servicio que consiste en una larga recta que desemboca en un acceso a la plazoleta que distribuye los coches entre nuestra dirección y la que lleva hacia Aluche.

Cuando vamos apretando el acelerador hacia dicha plazoleta, como humanos inteligentes que somos, sabemos que debemos decelerar convenientemente según nos acerquemos a la señal de ceda el paso dibujado en la plazoleta, pero en unas cuantas ocasiones la carretera se encuentra abarrotada de coches que provienen de la M-40 y que quieren abandonarla para buscar Boadilla y poblaciones similares.

En estos casos, nosotros solemos disminuir la velocidad no porque nos encontremos a la distancia adecuada del ceda el paso sino porque observamos que el coche de delante, si uno es previsor también el de delante a este, empieza a frenar. Nuestra conducta se acomoda a las condiciones del tráfico, pero realmente adecuamos la velocidad para no chocarnos con el de adelante, no porque deduzcamos la variable global de velocidad de ese grupo de coches que avanzan hacia la plazoleta.

Se puede pensar que nuestra inteligencia es limitada porque dependemos del conductor de adelante, que ese si que es el que lleva la inteligencia de ambos, pero no es así. Nosotros en nuestro coche, el conductor que vemos delante, el que este tiene delante y así hasta el primer coche parado en el ceda el paso esperando su oportunidad, somos escasamente inteligentes en este tipo de maniobras.

Reducimos nuestro potencial de inteligencia a una sencilla regla que es la ya comentada de disminuir la velocidad del coche hasta que se pare o se mantenga en una velocidad estable que nos posicione a una prudencial distancia del coche que vemos al mirar a la carretera o nos deje parados en el ceda el paso a la espera de una oportunidad para realizar el giro.

Para eso si que no es necesaria mucha inteligencia. Eso es perfectamente factible para un grupo de pájaros y, algunos de ellos es exactamente lo que hacen, de una forma sinceramente maravillosa.

Este tipo de comportamientos fueron estudiados por los doctores Eberhart y Kennedy en el año 1995, los cuales desarrollaron una técnica de optimización estocástica basada en poblaciones a la que dieron el nombre de Particle Swarm Optimization, o **PSO** en su acrónimo, que traducido literalmente significa Optimización en un enjambre de partículas.

Las técnicas de PSO son similares a las de los Algoritmos Genéticos, conocidos por su acrónimo **GA**, aunque difieren de estos en las características de mutación a través de los pasos evolutivos del algoritmo.

En el algoritmo genérico de PSO las posibles soluciones, llamadas partículas, sobrevuelan el espacio del problema siguiendo el recorrido realizado por las partículas de solución óptima en cada momento de ejecución del algoritmo.

Si continuamos la explicación utilizando nuestros pequeños pájaros hexagonales, podemos decir que cada pájaro sigue la ruta que mas le acerque a la mejor solución, comúnmente llamada **pbest** en el algoritmo.

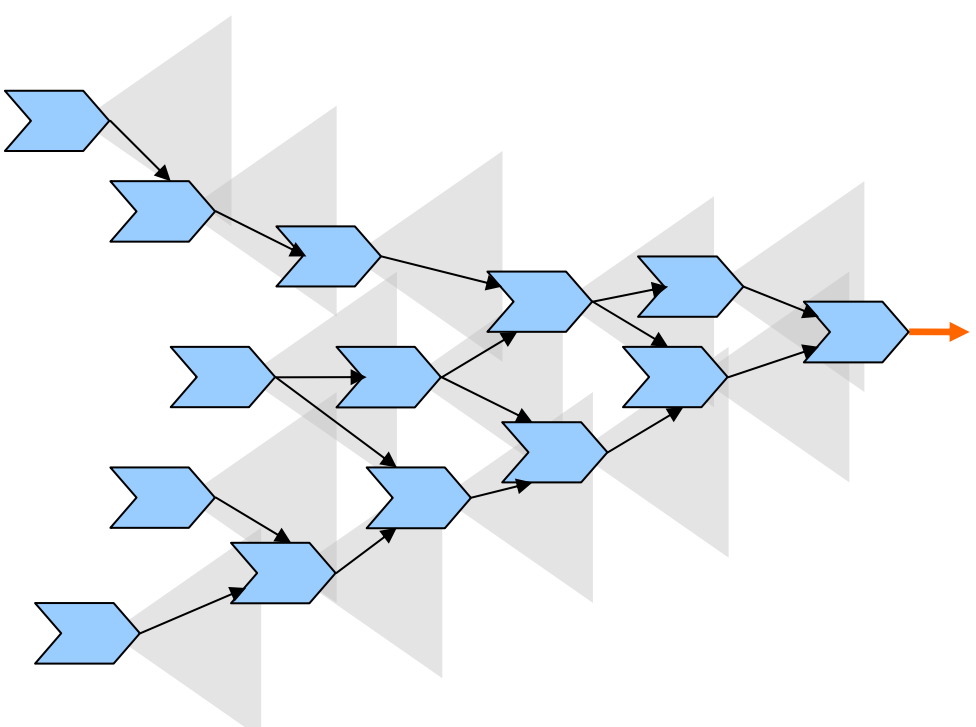
Además de tener en cuenta este valor, cada pájaro tiene en cuenta el mejor valor obtenido por sus vecinos de vuelo, que no son todos los pájaros del grupo sino aquellos que tiene más cercanos, un valor local a su entorno que recibe el nombre de **lbest** en el algoritmo.

En el supuesto hipotético de que pudiera conocer el valor local de sus vecinos, cuando estos vecinos supusieran la bandada entera, nuestro pájaro conocería un valor global que recibe el nombre de **gbest** en el algoritmo.

Pues el algoritmo de optimización lo que viene a requerir es que, en cada paso de ejecución, cada uno de nuestros pájaros modifiquen su velocidad en el vuelo, es decir que aceleren o deceleren maniobrando si es preciso, entre los valores que dispone para su pbest como para su lbest.

Si resumiéramos, se trata de hacer lo mismo que nosotros hacemos con nuestro coche al acercarnos a la fila de coches de la plazoleta.

Si lo viéramos como lo verían dentro de una bandada, los datos de interés para un pájaro vendrían a ser, en un esquema gráfico, los siguientes:



Como puede observarse en el gráfico, lo que todo pájaro debe tener en cuenta es la velocidad que lleva y un cálculo estimativo, dado por la experiencia que dan las horas de vuelo, de la posición y velocidad de aquellos vecinos visibles. Efectivamente, son vecinos los pájaros que le acompañan detrás, pero los pájaros no tienen retrovisor por lo que su entorno local es el compuesto por los pájaros dentro de un rango dado por el cono de visión de color gris claro del gráfico.

En función de su posición, su velocidad y las posiciones y velocidades locales que puede ver, calcula la aceleración adecuada para mantenerse en el grupo.

Si observamos en detalle los vectores de movimiento, dibujados con flechas en la gráfica anterior, veremos que cada pájaro tiene vectores de movimiento diferentes y, por tanto, resolverá su comportamiento con aceleraciones distintas para cada miembro del grupo. La única excepción se encuentra en el vector del pájaro guía que se mueve según un vector global que el resto de la bandada sigue.

Una forma de mayor distribución de la inteligencia, que elimina la existencia de lo que parece ser el pájaro mas listo del grupo, serían aquellas bandadas no dispuestas en forma de ala delta, típicas en las migraciones, sino un remolino aparentemente caótico formado por estas aves buscando no ir a terrenos cálidos sino comida.

El espacio de la comida, nuestro estimable plano en R^2 que pisamos cada día, es inmenso y la comida es escasa y de posición desconocida. Este caso es más interesante para nosotros ya que demuestra la eficacia de este tipo de algoritmos.

Lo que sucede con la búsqueda de comida tiene un principio desordenado. Nuestros pájaros sobrevuelan el entorno en direcciones erráticas, en busca de posiciones de comida. Muy probablemente la bandada en esos momentos se encuentre completamente dispersa. Cuando uno de los componentes de la bandada detecta comida, empieza a acercarse al espacio solución, una de ellas si hay varias zonas de comida, de forma que llega a posarse cerca del objetivo o en el objetivo mismo. Cuando los compañeros cercanos a este observan a su compañero posado, deducen que una de las partículas del sistema ha encontrado una solución al problema de la comida y empiezan a acercarse también a su solución, salvo aquellos cuyo campo visual no llega como para darse cuenta de su compañero posado. Lo mejor de este algoritmo es que estos lejanos pájaros si llegan a ver a compañeros suyos tendiendo hacia unas coordenadas de forma decidida y suponen que es mejor para su hambre seguir lo que parece una conducta de resolución del problema. Al fin y al cabo, la experiencia de siglos insertada genéticamente en su comportamiento les viene a decir que, aunque ellos no vean la solución, es mas próximo a la misma si siguen a aquellos que siguen a otros que a su vez siguen a algunos otros que siguen a ciertos miembros que han visto un compañero suyo comiendo comida.

El efecto en cadena de este algoritmo obtiene soluciones para todo el grupo que, de forma individual, serían imposibles. Además se obtienen de forma mucho más eficiente que si un solo pájaro intenta buscar comida en un vasto territorio.

Es tan sencillamente inteligente que sentí la necesidad de generar este tipo de modelo. Como viene siendo habitual, mis suposiciones cada vez mas ajustadas a la realidad de los obstáculos encontrados en el resto de tipos de modelo iban encaminadas pero encontraban una cierta resistencia.

Dicha resistencia venía dada por una sola razón. Para comprenderla veamos un algoritmo PSO genérico.

Este algoritmo tiene la siguiente forma:

Sea un espacio de solución multidimensional R^m *en el que se definen los siguientes elementos:*

- *Conjunto S con n partículas de la forma* $S = \{p_1, p_2, .., p_n\}$
- *Función objetivo f de la forma f:* $R^m \rightarrow R$
- *Cada partícula* p_i *de S tiene un vector de posición* $x_i = \{x_1, x_2, .., x_m\}$
- *Cada partícula* p_i *de S tiene un vector de velocidad* $v_i = \{v_1, v_2, .., v_m\}$
- *Sea* $\mathbb{X}_i$ *la mejor posición de una partícula* p_i *a la solución o pBest*
- *Sea* $\mathbb{G}$ *la mejor posición del conjunto S a la solución o gBest*

Se deben dar los siguientes pasos:

1. *Inicializar* $\mathbb{X}_i \leftarrow x_i$
2. *Inicializar* $\mathbb{G} \leftarrow MIN_{xi}\ [f(x_i)]$
3. *Mientras no se converja a una solución, hacer*
   - *3.1. Para cada partícula i:[1..n] hacer*
     - *3.1.1.* $x_i \leftarrow x_i + v_i$
     - *3.1.2.* $v_i \leftarrow \omega\ v_i + \varphi_1\ rand_1\ (\mathbb{X}_i - x_i) + \varphi_2\ rand_2\ (\mathbb{G} - x_i)$
     - *3.1.3. IF* $f(x_i) < f(\mathbb{X}_i)$ *THEN* $\mathbb{X}_i \leftarrow x_i$
     - *3.1.4. IF* $f(x_i) < f(\mathbb{G})$ *THEN* $\mathbb{G} \leftarrow x_i$
   - *3.2. GO TO 3.1 si i<n*
4. *GO TO 3 si no hay convergencia*

El algoritmo intenta la búsqueda de una solución óptima a través del trabajo acumulado de las partículas del conjunto.

Para ello, cada partícula se mueve a través del espacio de soluciones a una velocidad que modifica su posición y que se calcula mediante la suma de las siguientes expresiones:

- **$\omega\ \mathbf{v}_i$** que adecua la velocidad de la partícula según un factor inercial constante ω. En la práctica se suele tomar ω: [0.9,1).

- **$\varphi_1\ \mathbf{rand}_1\ (\mathbb{X}_i - \mathbf{x}_i)$** que actualiza la información de cada partícula de forma individual, es decir, ajusta la posición de la partícula en el espacio de soluciones en función de su mejor valor y su valor actual. Para esto se calcula el valor de la multiplicación φ_1 $rand_1$, donde φ_1 hace referencia a una constante del aprendizaje cognitivo de la partícula y $rand_1$ es un valor aleatorio. El valor obtenido se multiplica a cada componente ($\mathbb{X}_i$ - x_i) del vector diferencial entre la mejor posición de la partícula, $\mathbb{X}$, y su posición actual x.

- **$\varphi_2\ \mathbf{rand}_2\ (\mathbb{G} - \mathbf{x}_i)$** que actualiza la información de cada partícula dentro de la colectividad dada por el conjunto, es decir, ajusta el valor de la posición de la partícula en el espacio de soluciones en función del mejor valor del conjunto de partículas y su valor actual. Para conseguirlo, se calcula el valor de la multiplicación φ_2 $rand_2$, donde φ_2 hace referencia a una constante del conocimiento social de la partícula y $rand_2$ es un valor aleatorio. El valor obtenido se multiplica a cada componente ($\mathbb{G}$ - x_i) del vector diferencial entra la mejor posición del conjunto, $\mathbb{G}$, y su posición actual x.

Tras el cálculo de la nueva velocidad que tomará la partícula, echaremos un vistazo al valor de la función objetivo en la posición actual. Si el valor dado es menor, lo cual significa que es mejor, que el resultado de esta misma función en la mejor posición hasta el momento para la partícula, entonces la posición actual es tomada como la nueva mejor posición de dicha partícula o pBest.

Al igual que hemos hecho con la mejor posición, se deberá comprobar si la posición actual no es la mejor posición conocida, no ya de la partícula, sino del conjunto al completo. Si esto fuera así, entonces nuestro nuevo valor para gBest correspondería a la posición actual de esta partícula en concreto.

Este cálculo, realizado para cada una de las n partículas del conjunto, se iterará en un bucle cuya condición de parada será la obtención de una solución óptima, para el espacio de soluciones dado, mediante la función objetivo dada.

Una vez conocido en detalle el algoritmo PSO y antes de escribir un modelo inicial, deberemos tener en cuenta que la resistencia a su planteamiento no es sino sencillamente que se trata de un algoritmo. Y ya se comentó que el metasimulador extendido, por su estructura y su diseño, es capaz de abarcar muchos tipos de modelos, pero es especialmente sensible a aquellos que requieren de un algoritmo de ejecución.

No se está diciendo que no se pueda, se dice que se podrá solo si el metasimulador puede ajustar su funcionalidad para imitar la dada por el algoritmo.

Veamos si es posible.

Tenemos por una parte un bucle condicional **while** en el que existe un bucle incondicional **for** donde residen **asignaciones** y **condicionales**. Parece sencillo intuir que el metasimulador podrá realizar tanto las expresiones de asignación como los condicionales, pero pensemos que la primera fórmula hace referencia al cálculo de todos los componentes del vector de velocidad y la segunda hace referencia al cálculo de todos los componentes del vector de posición.

Bueno, es factible. Si consideramos una clase P para el conjunto de partículas, podemos escribir las expresiones anteriores de la siguiente forma:

Asignación $x_i \leftarrow x_i + v_i$

$$P.X = P.X + P.Vx$$
$$P.Y = P.Y + P.Vy$$
$$P.Z = P.Z + P.Vz$$

Asignación $v_i \leftarrow \omega\, v_i + \varphi_1\, rand_1\, (\mathbb{X}_i - x_i) + \varphi_2\, rand_2\, (\mathbb{G} - x_i)$

$$P.Vx = (w*P.Vx)+(c1*rnd()*(P.pBestX-P.X))+(c2*rnd()*(gBest-P.X))$$
$$P.Vy = (w*P.Vy)+(c1*rnd()*(P.pBestY-P.Y))+(c2*rnd()*(gBest-P.Y))$$
$$P.Vz = (w*P.Vz)+(c1*rnd()*(P.pBestZ-P.Z))+(c2*rnd()*(gBest-P.Z))$$

Asignación condicional IF $f(x_i) < f(\mathbb{X}_i)$ *THEN* $\mathbb{X}_i \leftarrow x_i$

$$P.F = Funcion(P.X,P.Y,P.Z)$$
$$P.FpBest = Funcion(P.pBestX,P.pBestY,P.pBestZ)$$

$$P.pBestX = if(P.F<P.FpBest,\ P.X,\ P.pBestX)$$
$$P.pBestY = if(P.F<P.FpBest,\ P.Y,\ P.pBestY)$$
$$P.pBestZ = if(P.F<P.FpBest,\ P.Z,\ P.pBestZ)$$

Asignación condicional IF $f(x_i) < f(\mathbb{G})$ *THEN* $\mathbb{G} \leftarrow x_i$

$$P.FgBest = Funcion(P.gBestX,P.gBestY,P.gBestZ)$$

$$P.gBestX = if(P.F<P.FgBest,\ P.X,\ P.gBestX)$$
$$P.gBestY = if(P.F<P.FgBest,\ P.Y,\ P.gBestY)$$
$$P.gBestZ = if(P.F<P.FgBest,\ P.Z,\ P.gBestZ)$$

Con lo que puede suponerse que el tipo de modelo que intentamos generar no nos lleva a ninguna imposibilidad. Al menos parece que sus constantes, variables y funciones son perfectamente posibles de modelizar.

El bucle for no es sino nuestro bucle principal de recorrido por la estructura de elementos instanciados en el espacio de simulación y como bucle while podemos utilizar la condición de parada del conjunto, por parada de cada una de sus partículas, o el uso de una función objetivo que no alcanza una solución concreta de forma que el modelo continua la ejecución hasta que el usuario pare la simulación en marcha.

El único escollo de mayor relevancia se encuentra en la asignación de un nuevo valor global para el conjunto de partículas, ya que si se cumple que el componente x_i del conjunto ofrece un valor $f(x_i)$, de su función de ajuste, inferior al valor $f(\mathbb{G})$ ofrecido por el actual valor global $\mathbb{G}$, entonces este valor se debe incorporar, justo en ese preciso momento, a todas las variables P.gBest del modelo.

Esto puede ser realizado gracias a la función extendida *Asigna(var, val)* que, como ya se ha comentado en la descripción de estas funciones, establece un valor de nuestra elección en la variable que indiquemos, en todos los elementos que posean esta variable. De esta forma la asignación es inmediata, tal como requiere el algoritmo original.

Por ello, el código de nuestro modelo prototipo modificará estas líneas de asignación de la siguiente manera:

Nueva Asignación condicional IF $f(x_i) < f(\mathbb{G})$ *THEN* $\mathbb{G} \leftarrow x_i$

```
P.FgBest = Funcion(P.gBestX,P.gBestY,P.gBestZ)

P.gBestX =
        if(P.F<P.FgBest,
                {Asigna(gBestX,{P.X})},
                {Asigna{gBestX,{P.gBestX})}
        )

P.gBestY =
        if(P.F<P.FgBest,
                {Asigna(gBestY,{P.Y})},
                {Asigna{gBestY,{P.gBestY})}
        )

P.gBestZ =
        if(P.F<P.FgBest,
                {Asigna(gBestZ,{P.Z})},
                {Asigna{gBestZ,{P.gBestZ})}
        )
```

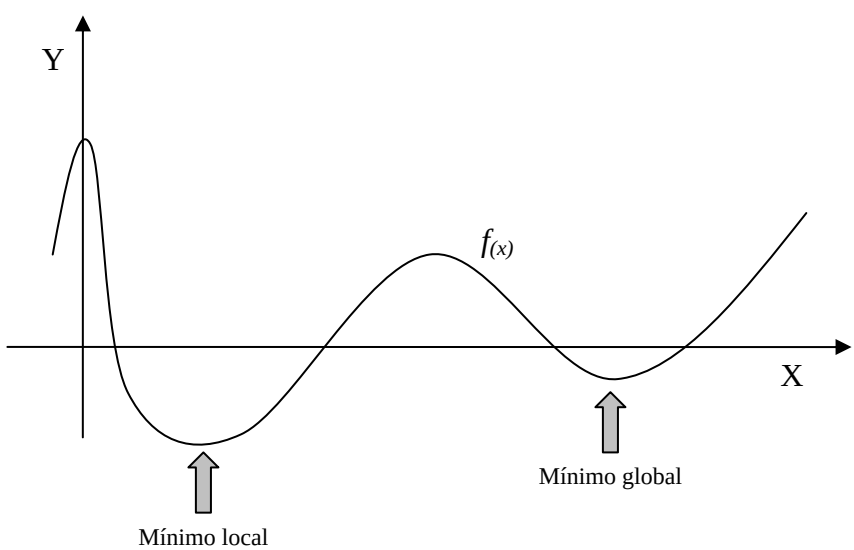
Una vez solventada la dificultad podemos ponernos a escribir nuestro primer ejemplo para este tipo de modelos. De hecho, una vez escrito el primer modelo para PSO podemos empezar a comprender mejor la teoría en la que se apoya.

Si, el algoritmo está ahí, transformado en líneas aceptables para nuestro metasimulador pero la ejecución del modelo nos indica a todas luces que la esencia de este algoritmo reside, con toda certeza empírica, en la elección de la función de ajuste para el problema tratado.

Si queremos que un enjambre de partículas obtenga una solución según los criterios PSO, deberemos estar dispuestos a encontrar una función *f* de ajuste, llamada *fitness function* en este algoritmo, que permita calcular valores particulares para cada elemento dado, valores para las mejores posiciones de estos elementos y valores para la mejor posición obtenida entre todas las posibles posiciones de los elementos en el espacio del problema. Pensar en una función adecuada es una tarea compleja si, además, se busca que nuestras partículas se comporten como pájaros revoloteando o en busca de comida.

Si lo pensamos detenidamente, los problemas de optimización mediante cálculo distribuido de partículas se basan en la localización de un mínimo óptimo, global en el espacio de soluciones.

Expresado en R^2 puede entenderse con mayor claridad:



Como puede observarse en la figura anterior, la función de ajuste $f_{(x)}$ elegida será la curva que describe el espacio del problema a resolver y podrá presentar varios puntos mínimos para la función. El sistema de partículas presentado se mueve en distintas regiones locales $[x_a,x_b]$ por la curva, memorizando en cada región aquellas posiciones donde el valor de *f* sea mínimo y detectando el mínimo de entre estos como el mínimo global de la curva. Tras iterar durante un número finito de pasos el algoritmo PSO, nuestras partículas llegarán a localizar la solución, el mínimo global existente, dirigiéndose en una especie de vuelo, modificación del punto de observación según una velocidad dada en cada paso que dependerá de la fórmula de velocidad vista anteriormente, hasta posarse en dicho mínimo.

Si ahora pensamos en nuestros pájaros en busca de un punto de comida situado en (x,y,0), algún punto de nuestro suelo, podremos elegir como función de ajuste la distancia en R^3, es decir, la función que cumple $d^2 = x^2 + y^2 + z^2$ y que nos servirá para localizar los mínimos valores de la misma, por cada uno de nuestros elementos, como mejor método de localización de la comida. Si un pájaro se mueve con el objetivo de minimizar la distancia que lo separa de la comida, terminará por cumplir nuestro objetivo.

Bajo este supuesto aparentemente infalible, es la función de cálculo de la velocidad la que destroza nuestras expectativas. Primero, la función de ajuste basada en la distancia entre dos puntos está viendo, para todo el rango posible dentro del espacio de simulación, los valores de la curva y no existe, como tal, una búsqueda de la solución, ya que la distancia nos da el mejor valor posible por cada punto.

Tan solo pueden mejorarse los mejores valores locales y el mejor valor global, si las partículas proceden a descender hacia el elemento que representa la comida. Pero es la fórmula general de la velocidad de la partícula la que nos sitúa a esta, en el siguiente instante de tiempo, fuera de su objetivo en magnitudes decenas de veces superior a lo que debería ser una aproximación adecuada.

El algoritmo prevé una restricción de la velocidad máxima de las partículas, de forma que puede imponerse un límite a la velocidad con la que los pájaros confluyen hacia la comida. Pero ni aún así.

Los elementos optan por la restricción de velocidad, que no es sino una ralentización en el sentido de movimiento pero de magnitud aleatoria, y dejan al sistema revoloteando en posiciones nada cercanas al objetivo.

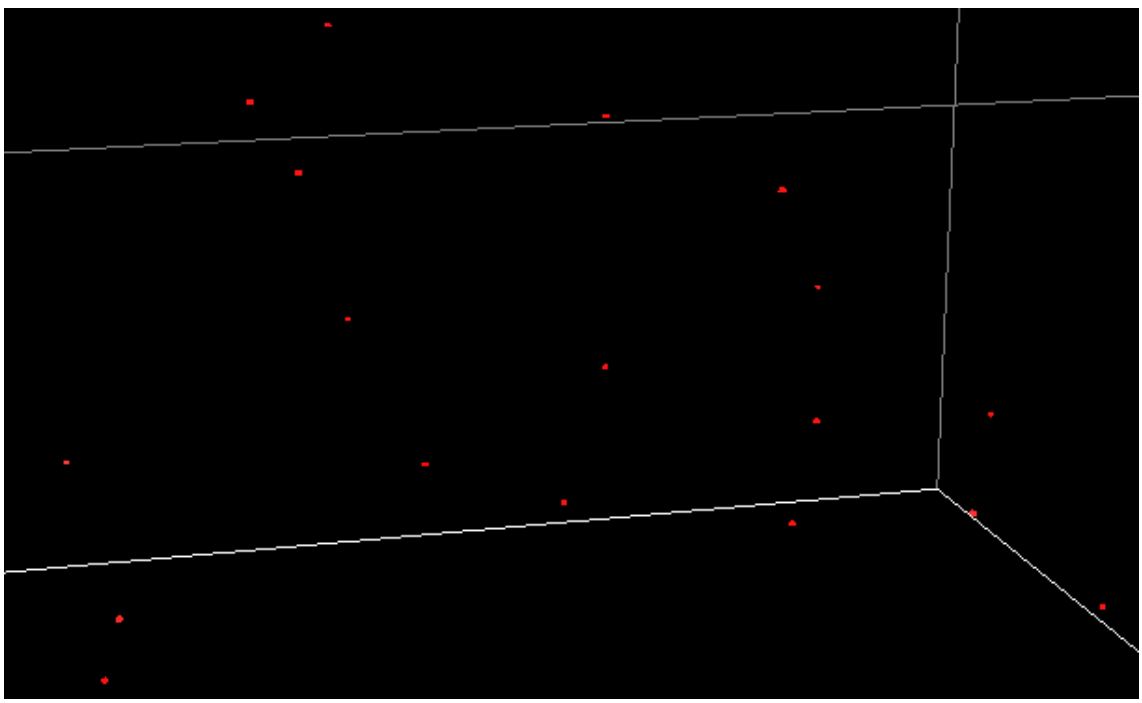
La conclusión, tras más de 12 modelos probados y 3 semanas de trabajo, es que las condiciones del metasimulador, un espacio fijo, una curva conocida en todo el rango dado y una función que conoce desde el primer momento la distancia al objetivo, hacen desaconsejable el uso puro del algoritmo PSO a estos casos.

Dado que el proyecto versa sobre un metasimulador, lo que si es un hecho es que la aplicación es capaz de representar el tipo de modelo y simularlo, aunque el modelo que estemos probando no logre satisfacer nuestros presupuestos al respecto de qué debería haberse ejecutado y de qué forma se deberían haber comportado los distintos elementos de la simulación.

Es decir, el metasimulador extendido no ejecuta modelos de forma errónea sino que simula modelos que, a veces, pueden ser conceptualmente erróneos. Debemos pensar en un fallo de nuestra idea y no en un error de ejecución del sistema.

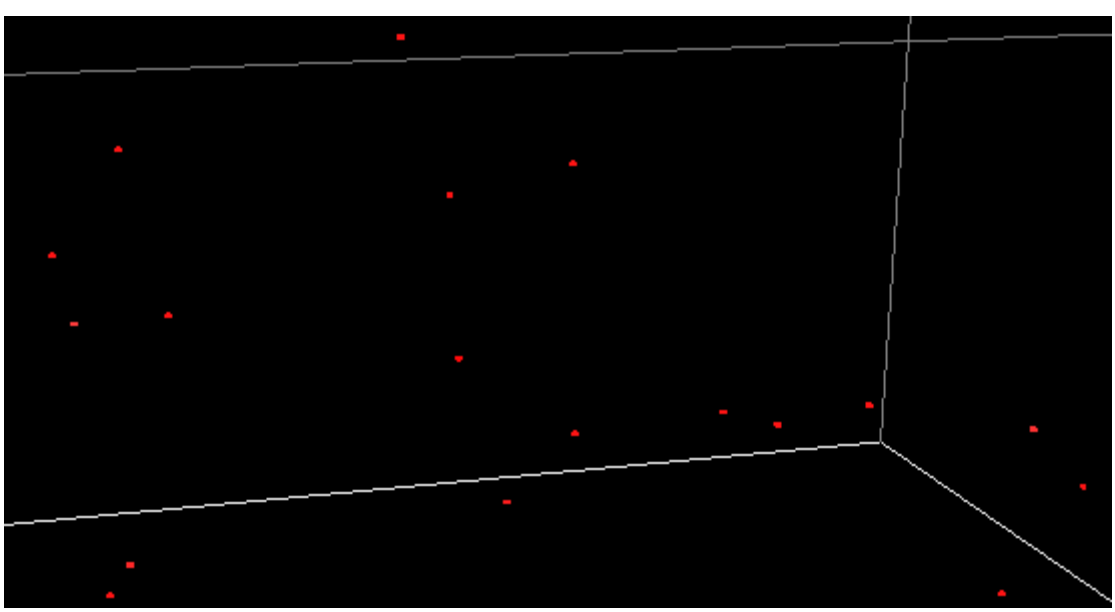
El simulador tan solo sigue al pie de la letra las órdenes que se le comunican a través del archivo de configuración, siendo nuestra responsabilidad asegurar que el diseño del modelo es correcto y, si lo es, va a simularse lo que deseamos.

Este es el caso, como puede comprobarse en la ejecución de **Swarm_1.conf** donde una serie de partículas buscan un valor óptimo que sea un mínimo global para la función dada:



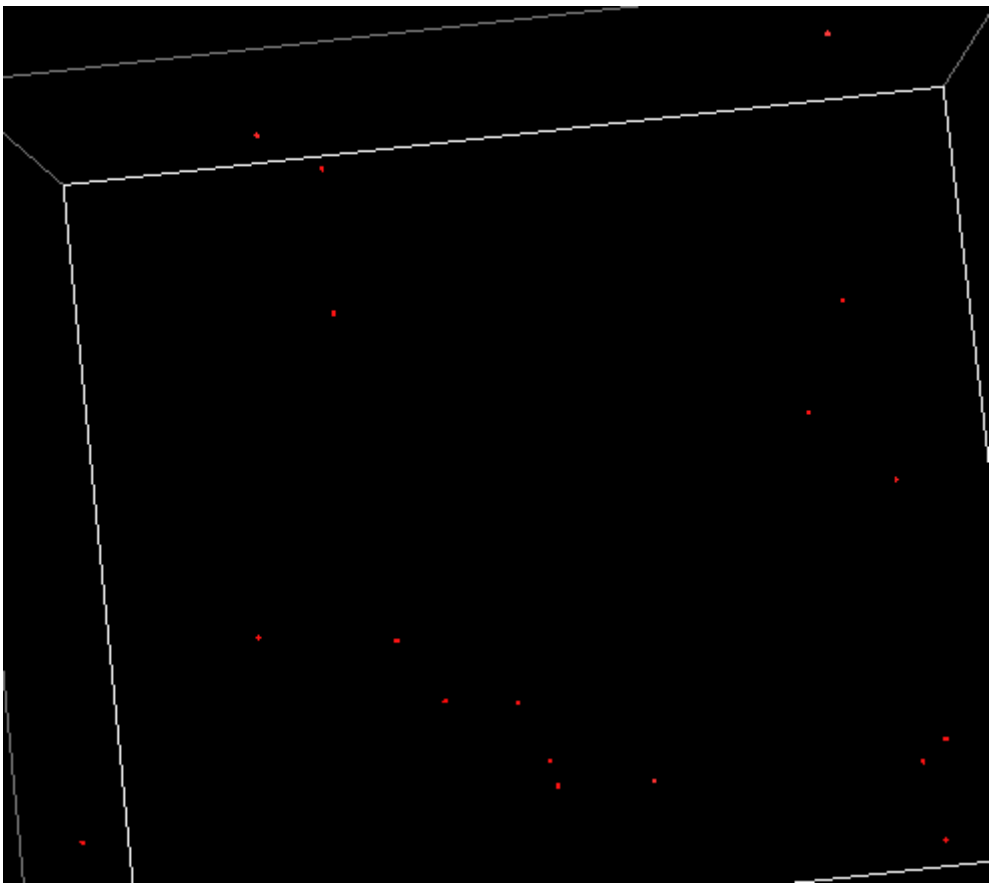
Modelo Swarm_1 en paso 0

El modelo dispone 18 partículas buscando una solución, con una *fitness function* de expresión $f_{(x,y,z)}=(x-100)^2+(y-100)^2+(z-100)^2$ que las permite alcanzarla, tras lo cual se vuelve a repetir el proceso sin solución de final.



Modelo Swarm_1 en paso 75

Las partículas empiezan a disponerse según una configuración que intenta optimizar la curva dada, aproximándose a los puntos mínimos, probablemente varios, de forma que se generan agrupaciones de partículas según la zona donde resida un mínimo del espacio de soluciones.



Modelo Swarm_1 en paso 2220

Si bien las partículas se ciñen a unas posiciones concretas, desde la óptica de la estética del modelo de cara al usuario, no podemos decir que sea visible la representación de una postura adoptada por el conjunto de partículas, no hay alineaciones espectaculares del conjunto, pero puede decirse que el simulador ejecuta según el algoritmo especificado.

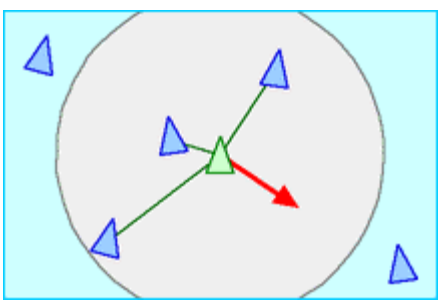
En cualquier caso, no, no son pájaros volando y uno desea verlos volar. Para ello, probablemente, deberá modificarse la elección de la función de ajuste y, en ese caso, deberemos retroceder de 1995 a 1986, desde Kennedy y Eberhart hasta uno de los personajes que inspiraron este algoritmo, Craig Reynolds, y su estudio de la cinemática de los pájaros en vuelo, conocidos como Boids.

Lo que Kennedy y Eberhart sintetizaron en una teoría formal sobre optimización basada en enjambres, provenía en su origen de una curiosidad por parte de Reynolds, al respecto de cómo simular en una computadora las interacciones entre pájaros en vuelo.

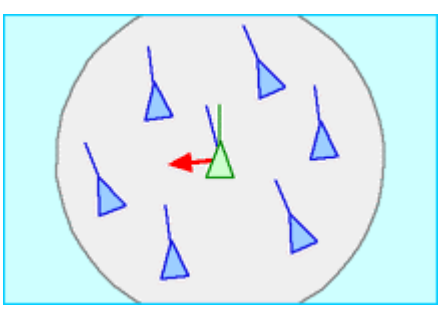
Existía una especie de comportamiento emergente entre los distintos sujetos implicados en una bandada que merecía un detenido estudio.

Efectivamente, tras un adecuado estudio de su comportamiento, Craig Reynolds llegó a la conclusión de que este sistema seguía una función de comportamiento óptimo que se basaba en las siguientes reglas:

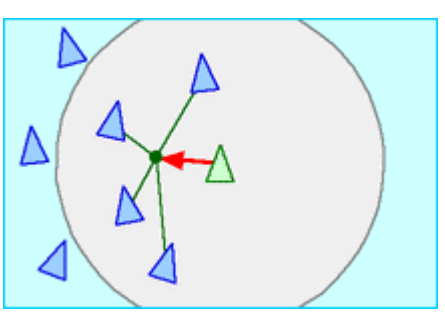
- Separación: Todo elemento del sistema debe maniobrar para mantener una distancia de separación con respecto a los elementos locales que le rodean.



- Alineamiento: Todo elemento del sistema debe volar en el sentido definido por el vector medio de los elementos locales que le rodean.



- Cohesión: Todo elemento del sistema debe volar de forma que se acerque al centro geométrico definido por el resto de elementos locales que le rodean.



Existen otras reglas que permiten afinar el comportamiento o llevarlo a comportamientos adecuados a la simulación que se pretenda en cada caso, pero estas son las tres reglas básicas que debe seguir todo pájaro que se precie de volar como parte de una bandada.

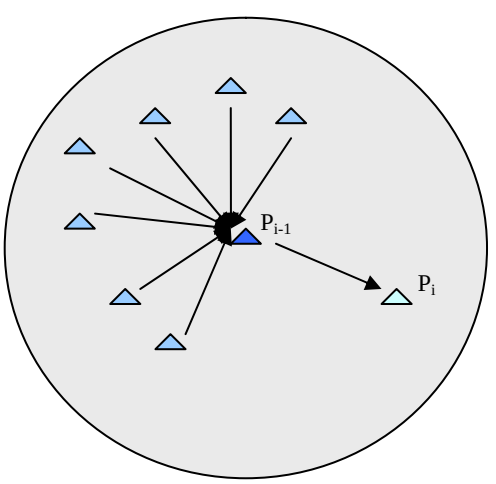
Al final, se tratará de adaptar la posición de cada elemento según un vector de velocidad que resultará de la combinación de las reglas anteriores.

Ahora, veamos si el metasimulador es capaz de plasmar en un modelo ejecutable el comportamiento del sistema comentado. Empezaremos por definir el círculo alrededor de cada pájaro y su relación con los vecinos de bandada.

Tal como explica el modelo original, cada pájaro controla una pequeña porción del espacio de vuelo, exactamente la que puede observar con su vista, por lo que el cono de visión define los vecinos de cualquier elemento. Son, sencillamente, los pájaros que puede ver. Por cuestiones de simplicidad, supondremos que su ángulo de visión es de 360º en todas las direcciones, de forma que podremos modelizar su entorno en una esfera de un radio determinado e igual para todos los elementos.

Dicho esto, un elemento situado en la posición P_{i-1}, por la primera de las reglas, deberá moverse a una posición P_i que será obtenida por la modificación de su posición actual y de un vector de repulsión, podríamos llamarlo, que tiene como objetivo único alejarse del resto de elementos visibles desde su posición. Cuanto mas cerca se encuentren los vecinos, mayor será la magnitud del vector de separación que lo aleje de los mismos.

Gráficamente, el vector puede describirse como sigue:



Matemáticamente se trata de obtener el conjunto de vecinos V, con todos los elementos dentro del círculo de visión del pájaro objeto de nuestro cálculo y observar cuales de ellos tendrían un vector vecino-pájaro con una magnitud inferior a una constante dada. En ese caso, este vector estaría provocando una reacción repulsiva en nuestro pájaro de sentido contrario al mismo.

La suma vectorial de aquellos vectores próximos al pájaro, nos proporcionaría la magnitud y el sentido del vector que debe ser aplicado a este.

En términos de nuestro pequeño lenguaje de simulación, podremos hacer uso de la función extendida Media() de la siguiente manera:

Sea P, uno de los pájaros en el modelo.
Sea k, un radio de proximidad

Calcular el punto medio de proximidad M de la forma:

Coordenada X de M como P.mpX = Media(P.k-X)
Coordenada Y de M, como P.mpY = Media(P.k-Y)
Coordenada Z de M, como P.mpZ = Media(P.k-Z)

Calcular el vector de separación Vs, como Vs = (M – P)

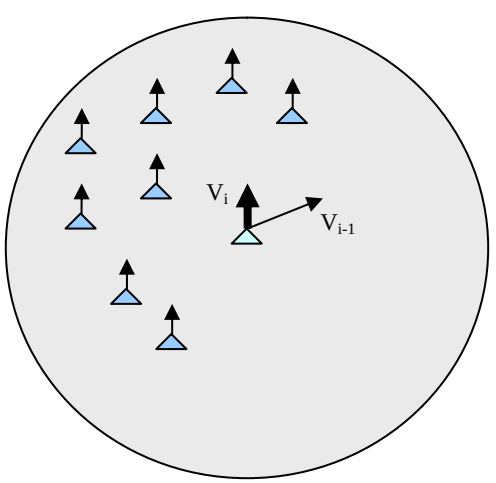
P.sepX = P.mpX - P.X
P.sepY = P.mpY - P.Y
P.sepZ = P.mpZ - P.Z

De esta forma obtendremos la nueva posición del elemento debida a la separación que debe realizar respecto de sus vecinos. Esta posición será una de las tres que deberán ser combinadas en un resultante del movimiento definitivo del elemento.

La segunda de las reglas atiende a igualar el vector de vuelo del elemento a tratar con el vector medio calculado mediante la media de los vectores de cada uno de los elementos vecinos a este.

De esta forma conseguimos que nuestro pájaro vuele acorde a la dirección y sentido del resto de los elementos y en condiciones estacionarias de vuelo, al resto de la bandada.

Gráficamente, la alineación podría describirse de la siguiente forma:



En otras palabras, debe adecuarse el vector al vector medio local.

Para conseguir esto, deberemos calcular el vector medio de los vecinos a P, que será el valor resultante de calcular la media de las magnitudes de cada vector de desplazamiento de un pájaro vecino a P, de la siguiente forma:

Sea P, uno de los pájaros en el modelo.
Sea k, un radio de proximidad

Calcular el vector medio Vm de la forma:

Coordenada X de Vm como P.vmX = Media(P.k-Vx)
Coordenada Y de Vm, como P.vmY = Media(P.k-Vy)
Coordenada Z de Vm, como P.vmZ = Media(P.k-Vz)

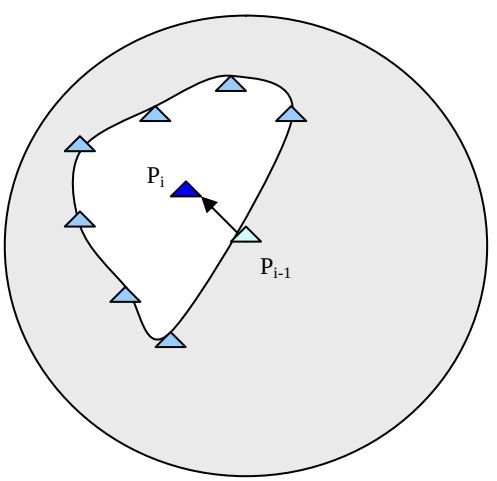
Calcular el vector de alineación Va, como Va = Vm - P

P.aliX = P.vmX - P.X
P.aliY = P.vmY - P.Y
P.aliZ = P.vmZ - P.Z

Nuestro pájaro tendrá ya un sentido de vuelo adecuado al de sus vecinos.

La tercera de las reglas anima a cada elemento a situarse en el grupo local, en caso de que se encuentre ligeramente separado. Para ello calculará la distancia existente a los elementos vecinos y si es superior a una constante dada, adecuará la velocidad para que su posición sea modificada para integrarse con el grupo.

Gráficamente, la cohesión puede definirse de la siguiente forma:



Nuestro pájaro intentará aproximarse a una posición cercana a la posición media calculada utilizando sus vecinos locales.

Para ello, de forma similar a la primera regla, la tercera regla requerirá el cálculo de la posición media del grupo de vecinos. Esta media y el cálculo de aproximación a la misma se harán de la siguiente forma:

Sea P, uno de los pájaros en el modelo.

Sea k, un radio de proximidad

Calcular la media de cohesión Pm de la forma:

Coordenada X de Pm como P.cmX = Media(P.k-X)

Coordenada Y de Pm, como P.cmY = Media(P.k-Y)

Coordenada Z de Pm, como P.cmZ = Media(P.k-Z)

Calcular el vector de cohesión Vc, como Vc = Pm - P

P.cohX = P.cmX - P.X

P.cohY = P.cmY - P.Y

P.cohZ = P.cmZ - P.Z

Una vez disponemos de los valores para cada una de las reglas, el vector de velocidad resultante que modificará la posición de nuestro pájaro será la combinación de estos tres valores calculados:

Calcular el vector resultante V como V = Vs + Va + Vc:

P.Vx = P.sepX + P.aliX + P.cohX

P.Vy = P.sepY + P.aliY + P.cohY

P.Vz = P.sepZ + PaliZ + P.cohZ

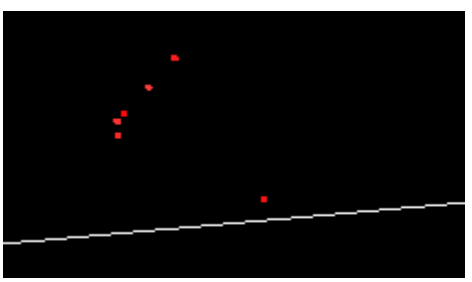
y aplicado sobre la posición del elemento modificará su valor dentro de los requisitos de estas tres reglas, con lo que nuestro pájaro se mantendrá acorde al comportamiento de sus vecinos de bandada.

Tenemos lo necesario para escribir un nuevo archivo de configuración, llamado **Swarm_2.conf** que incluirá los cálculos anteriores y una regla extra que incorpora un comportamiento para evitar que los pájaros queden estacionados en el punto (X_{MAX},Y_{MAX},Z_{MAX}) del espacio de representación, al no tener la bandada ningún objetivo concreto de vuelo como podría ser la captura de una presa o huir de algún depredador en la zona.

Este modelo, dispone de un número pequeño de elementos situados en la zona media del espacio de representación con una velocidad inicial de una posición por tic de tiempo.

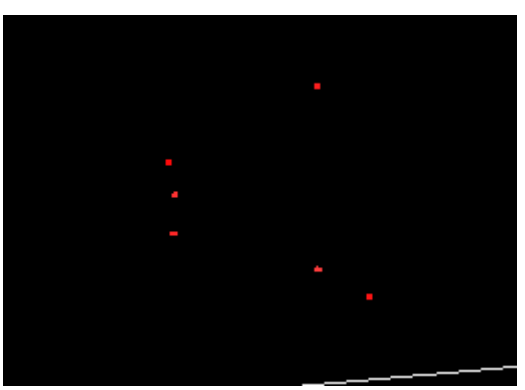
Modelo Swarm_2 en paso 0

Cuando iniciamos la ejecución cada elemento calcula el vector de desplazamiento adecuado que le mantenga coordinado con respecto a sus vecinos, que en el caso del ejemplo son todos o la inmensa mayoría. En los primeros pasos podemos observar como todos los elementos comienzan a dirigirse a valores positivos superiores de los tres ejes de coordenadas, a excepción de uno de ellos, debido a que su radio de visión no alcanza a ningún vecino con el que adecuar sus movimientos, por lo que continúa volando con total independencia del grupo.



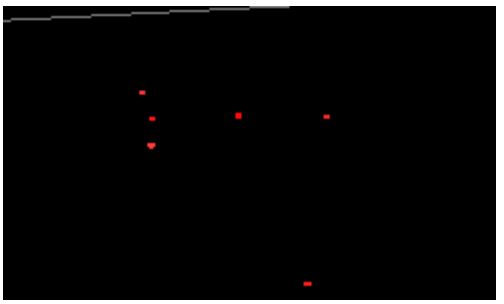
Modelo Swarm_2 en paso 3

A partir del cuarto paso comienza a aproximarse a alguno de los elementos restantes.



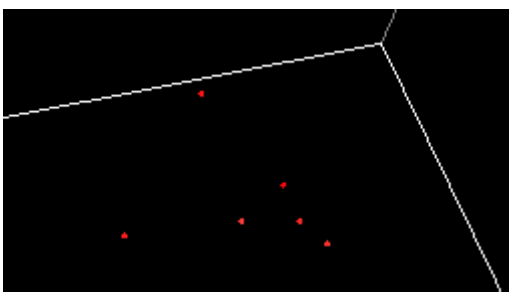
Modelo Swarm_2 en paso 8

Unos cuantos movimientos mas adelante, la disposición del grupo sigue sin ser completa, faltando nuestro famoso rezagado que vuelve a perder contacto con algún vecino con el que adecuar su comportamiento.



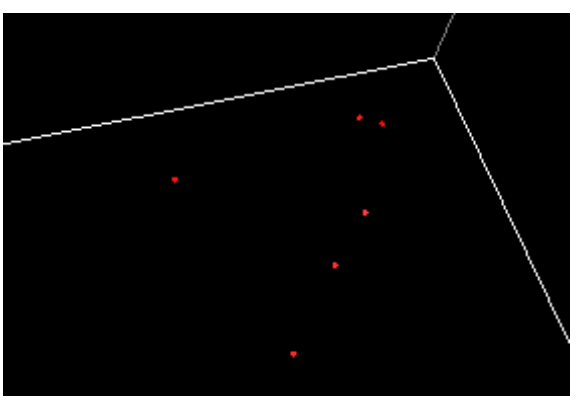
Modelo Swarm_2 en paso 24

A partir de este momento, el resto de la bandada comienza a aproximarse al límite permitido del espacio de representación y actúa contra ellos la cuarta regla de distancia a las paredes del espacio. Una y otra vez, cada pájaro que llega al límite establecido de proximidad con la pared, huye de la misma casi repelido por una fuerza contraria a su vector resultante, de forma que destruye la armonía del grupo, que debe reorganizarse.



Modelo Swarm_2 en paso 187

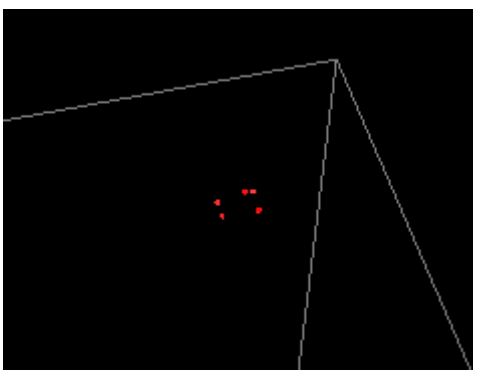
Si continuamos la ejecución, nada de lo anterior cambiará. Todos los elementos vuelan para mantenerse en el grupo o se alejan con precipitación cuando se aproximan al límite.



Modelo Swarm_2 en paso 345

Durante los instantes en los que la simulación corresponde a un vuelo libre, podremos apreciar al emergencia en el sistema de un comportamiento de grupo que podrían mantener sin descanso, salvo por la inclusión de la regla de colisión contra los límites. Es cierto que esta regla los intenta dispersar y el mismo comportamiento al que nos referimos termina por devolverlos a una disposición de bandada, acorde a las funciones del modelo dado.

Si intentamos disponer un modelo que mejore la regla de la repulsión podremos obtener un resultado bastante similar al del comportamiento de un enjambre de partículas. Así, en el modelo **Swarm_3.conf**, se ha modificado la regla del límite del espacio de representación para que, en vez de verse repelidos en sentido opuesto pero con magnitud aleatoria, varíen su vector de movimiento en diez unidades en el sentido de alejamiento, una vez la posición llegara a traspasar cualquiera de los límites dados por dicho espacio.



Modelo Swarm_3 en paso 1536

Como puede observarse, al ser la fuerza de repulsión constante y menor en magnitud a la del modelo anterior, los elementos casi no deben hacer esfuerzo por reagruparse, sino por mantenerse a una mínima distancia de cada uno de sus compañeros, lo que nos deja un grupo aparentemente estacionario sobre posiciones cercanas a la esquina diagonal superior del modelo.
Con todo, sigue sin apaciguarse la intención de generar una auténtica bandada de pájaros o, mejor dicho, un grupo de elementos de simulación que se muevan como si de una bandada se tratara.

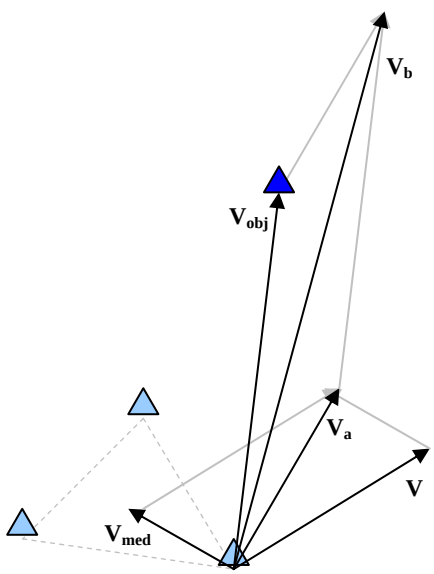
Si pensamos detenidamente en el comportamiento resultante de los dos modelos anteriores, concluiremos que el sistema tiende a buscar una solución adecuada a la ecuación general de suma de vectores medios de los grupos locales del enjambre.

Esto es correcto y ambos modelos funcionan adecuadamente, si no nos importa que su vuelo vaya desde las posiciones de partida hasta algún momento en la ejecución en que podríamos parar el sistema por recurrente. Esto, que nos debe recordar mucho a los problemas comentados con el primero de los ejemplos que se basaba únicamente en el algoritmo PSO existente, tiene una sencilla variante, recomendada por todos los expertos en juegos, secuencias de ordenador y modelos cinemáticas en general.

Necesitamos darle un objetivo al que seguir y nuestra bandada dejará de tener un vuelo errático como única meta de su ejecución.

Nuestro nuevo ejemplo, llamado **Swarm_4.conf**, se basa en cálculos realizados por cada elemento en relación a un líder cuya trayectoria de vuelo recorre todo el espacio de simulación, de forma que cada elemento simple calcula el vector de desplazamiento al punto donde el líder se encuentra antes de la ejecución de un paso y donde se encuentra tras la ejecución de dicho paso.

Podemos tener una idea mas clara con el siguiente gráfico:

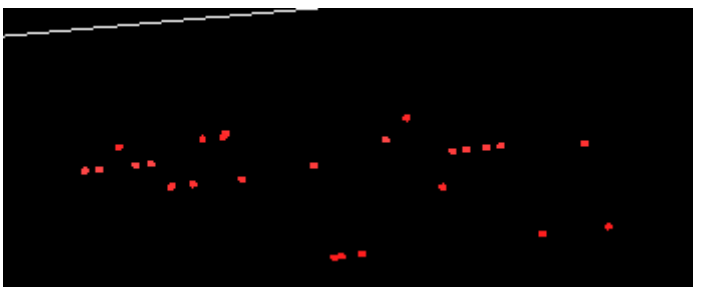


Como puede observarse en la imagen anterior, cada elemento dispone de un vector inicial V y puede calcular el vector medio con respecto a los vecinos locales V_{med}, de forma que puede obtener un vector V_a que es la suma de ambos.

Dado un vector al objetivo en movimiento, V_{obj}, puede sumar el anterior vector V_a a este y obtener un vector resultante V_b que es el vector que asumirá como nuevo vector de desplazamiento para el elemento en el siguiente paso de ejecución.

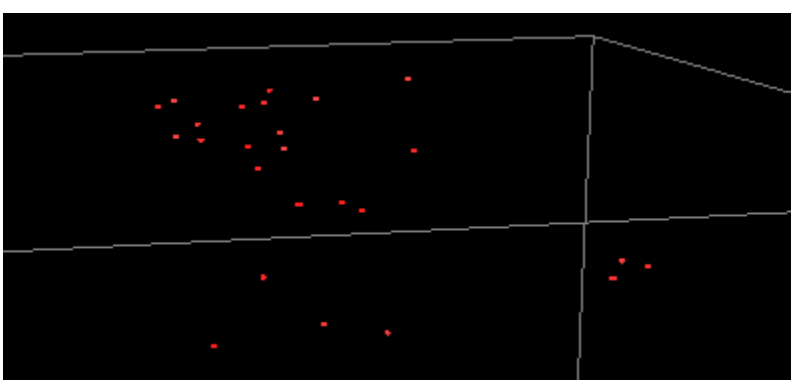
Si nuestro líder se construye como un elemento imaginario que se mueve según una serie de funciones globales al sistema, estaremos en condiciones de hacer que los pájaros de la bandada sigan a este señuelo y presenten un comportamiento homogéneo como grupo.

Si observamos la ejecución de este modelo veremos que exhibe, de forma constante, un vuelo en bandada que en su máxima velocidad de representación se asemeja a un enjambre de partículas.



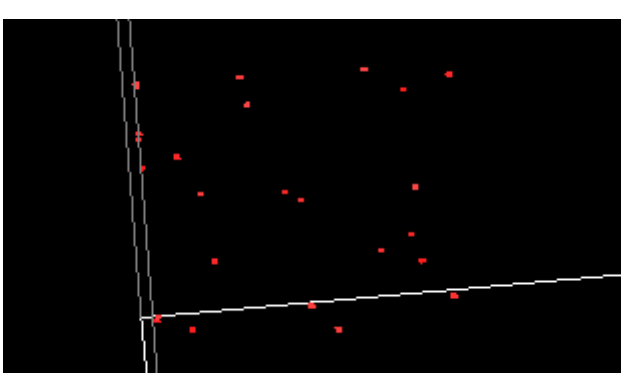
Modelo Swarm_4 en paso 0

Los primeros movimientos alinean cada partícula en busca del objetivo en movimiento o, si se prefiere ver desde ese punto de vista, dotan a cada elemento del enjambre de una trayectoria que deben intentar seguir.

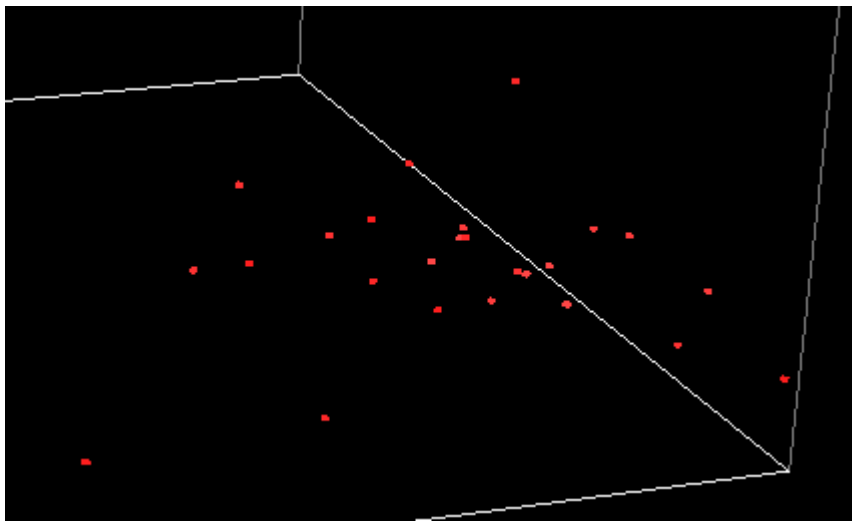


Modelo Swarm_4 en paso 21

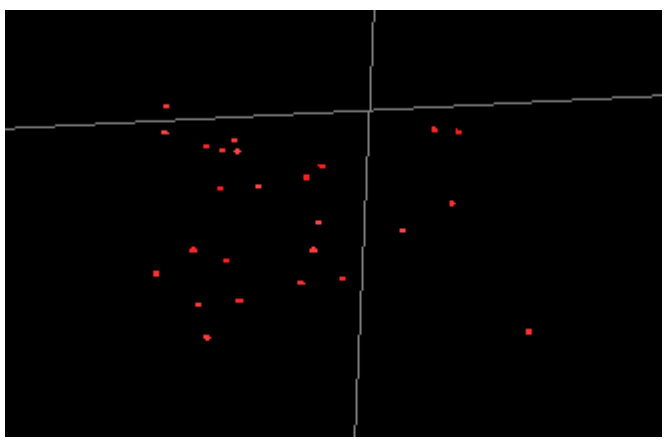
A partir de ese momento no hacen sino seguir a este líder a través del espacio de representación como una bandada ávida por capturar a su presa.



Modelo Swarm_4 en paso 93



Modelo Swarm_4 en paso 159



Modelo Swarm_4 en paso 186

Este modelo se ejecutará sin fin, manteniendo un bucle de vuelo en busca del objetivo. Por fin una bandada de pájaros para satisfacer las expectativas existentes en el inicio de este apartado.

13 MODELOS TIPO V

IFS y Chaos Game
Sierpinski, Barnsley y el árbol de la manzana

Para iniciar este tipo de modelos, recurrí a las hojas garabateadas en una de las reuniones con mi tutor, ya que en ellas se encontraba uno de los modelos que mas me atrajeron al inicio de este proyecto.

Como ya he dicho en apartados anteriores, todo comenzó por el interés en poder describir las funciones de crecimiento de un coral. Este interés para la generación de corales lo compartían en igualdad de condiciones tanto esta familia como la de los líquenes y la de los helechos. Y por estos últimos y los garabatos en el papel terminé por conducir mis pasos hacia el interior del mundo de los fractales.

Un fractal consiste en una figura semi-geométrica que tiene una forma básica que aparece repetida en distintas escalas de observación. En 1975, el matemático Benoît Mandelbrot utilizó el término latín *fractus*, cuyo significado es *quebrado* o *fracturado* para bautizar estas estructuras.

Los fractales pueden obtenerse mediante un proceso iterativo o recursivo que se encarga de reproducir la estructura inicial como estructuras autosimilares a cualquier escala a la que nos situemos en el conjunto.

Este hecho convierte a los fractales en estructuras de geometría irregular y llenas de detalle en un rango infinito de escala. Uno de los paseos más emocionantes que puedan darse es a través del conjunto de Mandelbrot en 3D, una de las estructuras famosas en este extraño mundo.

Tanto como avancemos nuestra imaginaria nave a través de su espacio seguiremos encontrándonos ante nosotros la repetición de las pautas geométricas que lo constituyen.

Es como si descendiéramos desde el espacio hasta el planeta Tierra y comprobáramos como una y otra vez el fractal natural que supone la línea divisoria entre la tierra y el mar no dejara de hacerse patente.

La gran diferencia estriba en que el conjunto de Mandelbrot no se topa con ningún límite y sigue de forma infinita su expansión.

Las características comunes a todo fractal, tal como son reunidas por Kenneth Falconer en su obra *Fractal Geometry: Mathematical Foundations and Applications*, son las siguientes:

- No puede ser descrito en términos geométricos tradicionales

  Es evidente que, por ejemplo, un fractal natural como pueda ser el que existe en los copos de nieve que caen en invierno no tenga una descripción geométrica como la existente para un triángulo, un hexaedro o un paraboloide.

- Posee detalle a cualquier escala a la que se observe

  Al contrario que cuando nos aproximamos a un cubo o a un dodecaedro, un fractal sigue presentando detalles de su estructura básica por muy cerca que estemos de su límite aparente.

- Es exacta o estadísticamente autosimilar

  Todo fractal es similar a sí mismo de forma recursiva.

  Si nos fijamos en las líneas de una costa, veremos que difieren poco de las líneas geográficas de la frontera de un extenso país y estas, de las líneas geográficas de los continentes vistas a escala planetaria.

- La dimensión de Hausdorff-Besicovitch es mayor que la dimensión topológica

  La dimensión de Hausdorff-Besicovitch, una de las muchas definiciones para el término de dimensión matemática, se obtiene como el punto resultante del punto de inflexión del valor de la potencia elegida en la longitud de Hausdorff, que no es sino la suma del diámetro topológico elevado a una potencia s de un recubrimiento entero del objeto a partir de entornos o cubrimientos de diámetro $d \leq \Delta$ del propio objeto, cuando dicha longitud pasa de ser infinita a nula.

- La definición puede ser realizada mediante un sencillo algoritmo recursivo

  Dado que no podemos definir un fractal en términos geométricos pero queremos utilizarlos en nuestras aplicaciones, deberemos definirlos mediante alguna herramienta matemática.

  La mejor definición posible de un fractal viene dada por su algoritmo recursivo.

Una vez definido el término de fractal veamos de qué forma pueden ser generados, para emplear estas técnicas en nuestros modelos para este tipo de simulación.

Un método de construcción fractal es el denominado como **Sistemas de Funciones Iteradas** que proviene del término inglés **I**terated **F**unction **S**ystems, cuyo acrónimo es **IFS**.

Este método, desarrollado por el inglés John Hutchinson, genera fractales mediante la unión de distintas copias de sí mismos, donde cada copia es transformada por una función, en los modelos más sencillos, o, por un sistema de funciones de transformación, en aquellos modelos que requieren de distintas operaciones de rotación y desplazamiento sobre el conjunto original que provee la base para la generación del fractal buscado.

Mas adelante veremos ejemplos de simulación para generar tanto uno como otro tipo de fractales.

La definición matemática del método IFS es como sigue:

$S = U_{i=1..N}\ [f_i(S)]$ *tal que:*

S es punto fijo de un operador de Hutchinson

S es un subconjunto de $\mathbb{R}^n$

$f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ *es el conjunto de funciones a iterar*

S representa la unión de las distintas funciones

Nuestro método IFS es todavía demasiado matemático para ser escrito como modelo en el metasimulador, por lo que incluiremos ahora la segunda de las herramientas a utilizar, conocida como el juego del caos o, **Chaos Game**.

El juego del caos es uno de los algoritmos más comunes para generar fractales IFS y se basa en la elección de un punto aleatorio en el plano sobre el que se aplica una de las distintas funciones del sistema, según un valor de probabilidad establecido.

Veamos nuestro primer modelo, llamado **Sierpinski.conf** que, como nos adelanta su nombre, se trata de una variación del **Triángulo de Sierpinski** que nos muestra lo que podríamos llamar una **Pirámide de Sierpinski** en las tres dimensiones.

El modelo sigue el método IFS de construcción para la figura del triángulo de Sierpinski pero añade una tercera componente para generar la pirámide mencionada.

Básicamente, el modelo consta de lo siguiente:

Variables declaradas

[CLASE_VAR]

A.id=-1

A.Sel=0

A.elm=-1

Funciones

[CLASE_FUNC]

A.Sel=if(A.id=={Fichas()}-1,1,0)

*A.elm=if(A.Sel==1,{Entero(4*rand())},-1)*

Inserción de nuevas fichas

[CLASE_INS]

(A.id==-1&&T==2)
A.X=({Ficha(0,X)}+{Ficha(1,X)})/2

A.Y=({Ficha(0,Y)}+{Ficha(1,Y)})/2

A.Z=({Ficha(0,Z)}+{Ficha(1,Z)})/2

A.R=1

A.id={Fichas()}-1

*A.elm={Entero(4*rand())}*

A.Sel=1

(A.Sel==1&&T>2)
A.X=(Fpid.X+{Ficha({Fpid.elm},X)})/2

A.Y=(Fpid.Y+{Ficha({Fpid.elm},Y)})/2

A.Z=(Fpid.Z+{Ficha({Fpid.elm},Z)})/2

A.R=1

A.id={Fichas()}-1

*A.elm={Entero(4*rand())}*

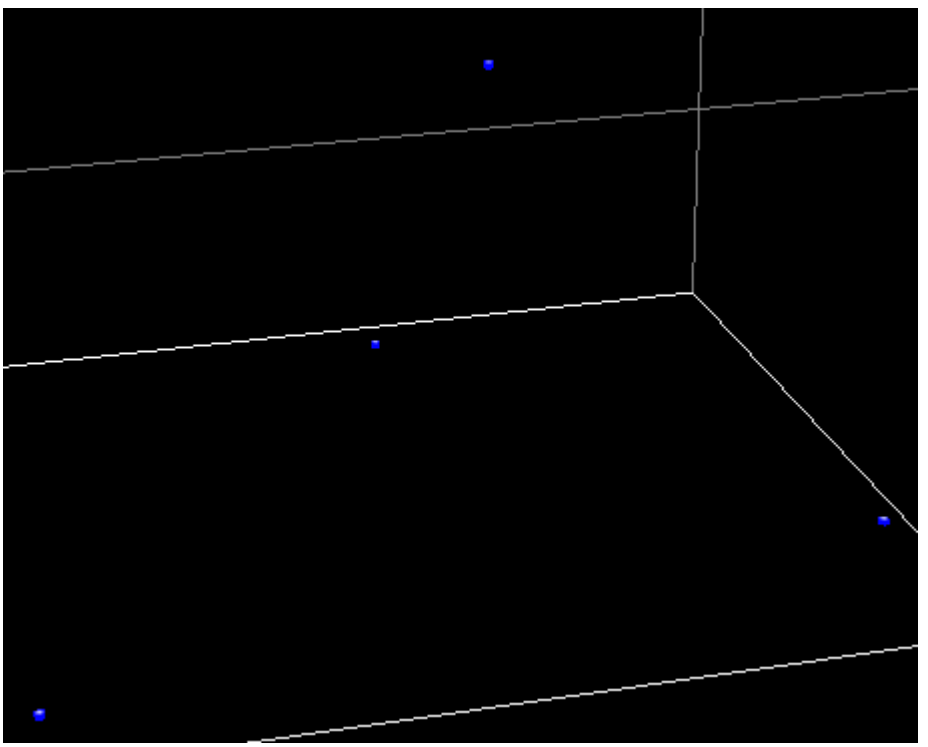
A.Sel=1

En resumen, el modelo elije en cada paso de ejecución una ficha entre las existentes y traza una recta entre esta y una de las cuatro fichas iniciales de la pirámide. Tras esto, calcula el punto medio de la recta trazada y crea en dicho punto una nueva ficha para la pirámide generada.

Este proceso se ejecuta indefinidamente y, para mejorar la velocidad de computación, el modelo deja paradas todas las fichas no elegidas de forma que el cálculo se centra tan solo en el cálculo del punto medio del segmento formado por el punto seleccionado y uno de los puntos iniciales, elegido al azar, en cada paso de la simulación.

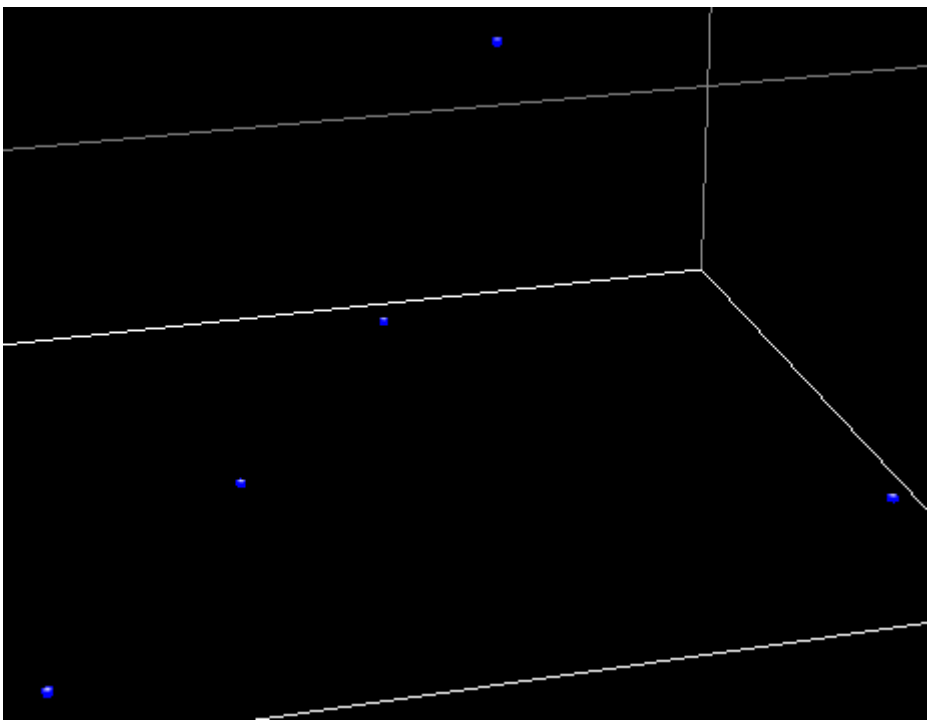
Todo lo demás es trabajo del sistema de funciones, en este caso una sola, que al ser afines, nos devuelven siempre un punto dentro del espacio de soluciones S, situado en posiciones aleatorias pero que terminan por cubrir el modelo solución.

La ejecución de este modelo parte de cuatro puntos que delimitan los límites de la pirámide.



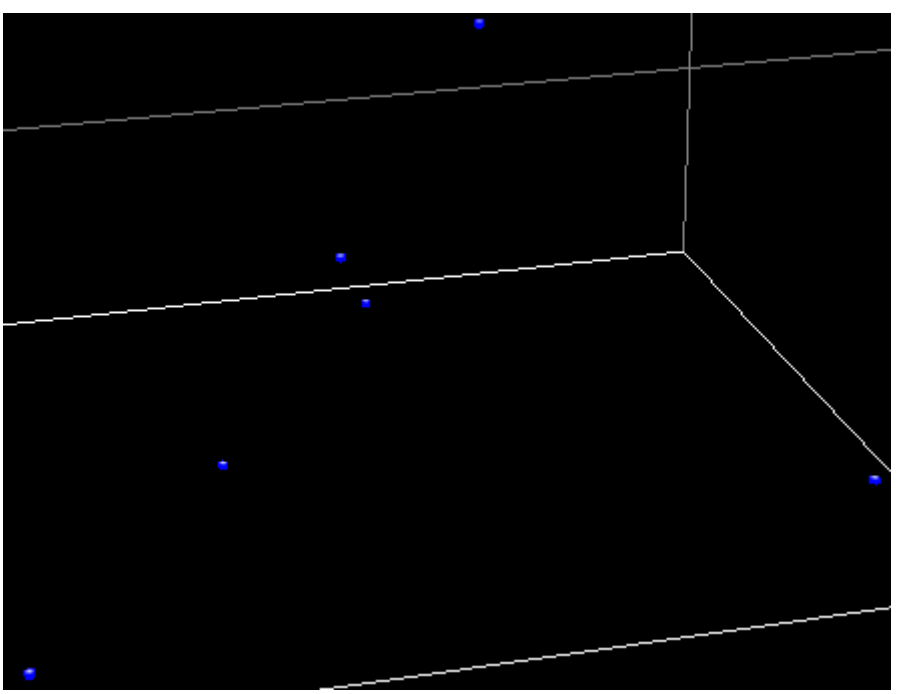
Modelo Sierpinski en paso 0

Se toman dos puntos y se inserta un quinto en la mitad del segmento definido por estos.



Modelo Sierpinski en paso 1

Se toma el quinto punto y uno de los vértices y se realiza una nueva inserción.

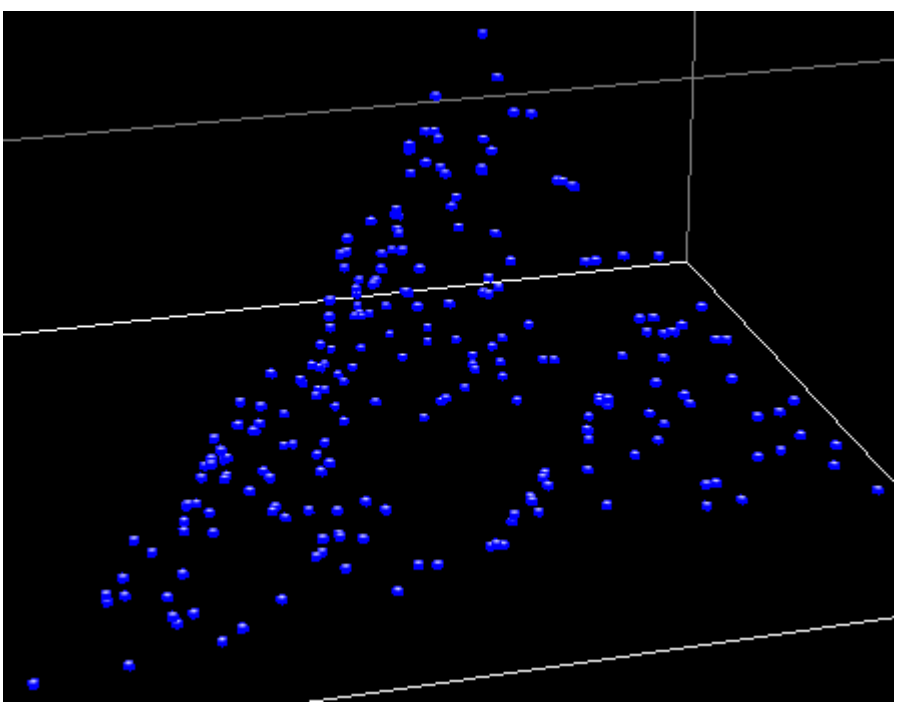


Modelo Sierpinski en paso 2

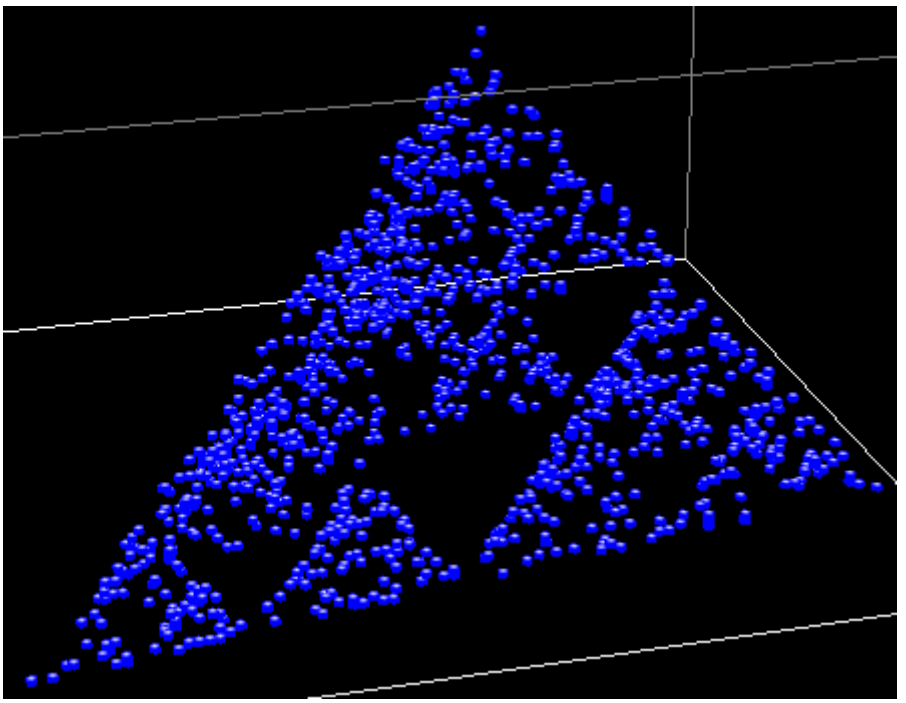
Este proceso se repite una y otra vez, tomando el último punto insertado y trazando un segmento desde este punto a uno de los cuatro puntos iniciales, los vértices de la pirámide, insertando en la mitad del segmento el nuevo punto del algoritmo.

Tras unos cuantos cientos de iteraciones, el modelo empieza a asemejarse a una pirámide llena de huecos y basada en pequeñas construcciones de pirámides similares a la de partida. Si la pirámide inicial es de base equilátera o el segmento del centro geométrico de su base a su vértice mas elevado se encuentra inclinado 30º respecto a la normal, las pequeñas pirámides que completan nuestra pirámide inicial, tendrán una base equilátera y su segmento de centro de la base a su vértice mas alto inclinado exactamente 30º sobre la normal, como la pirámide de partida.

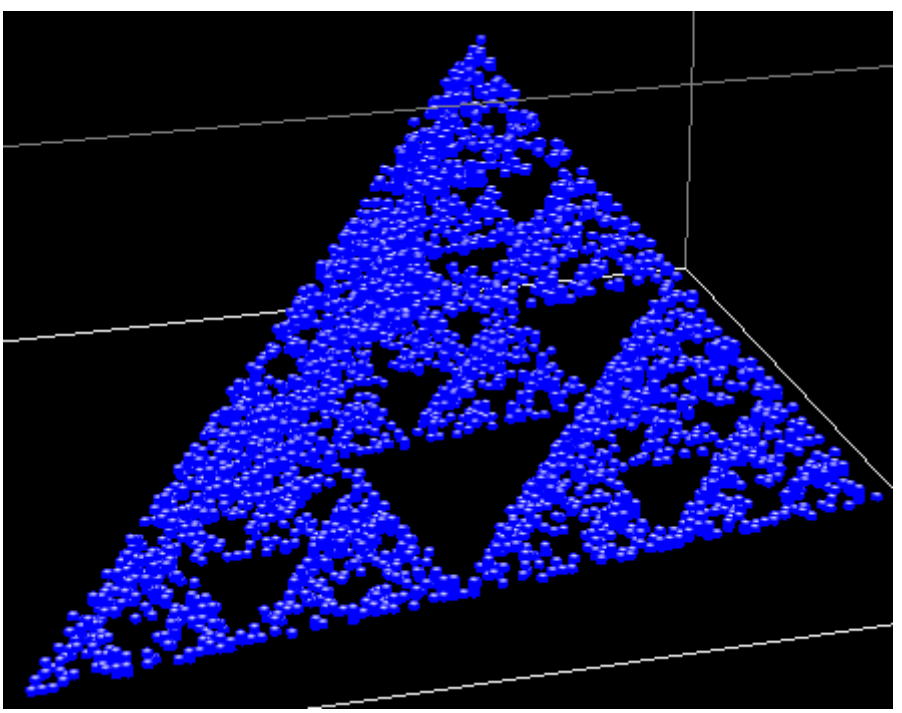
Esto es lógico, pues se trata de figuras que repiten una estructura base de forma autosimilar o, visto desde el otro lado, son autosimilares, porque encontramos la misma estructura básica en la pirámide de partida o en cualquiera de las pequeñas pirámides en las que nos fijemos.



Modelo Sierpinski en paso 231



Modelo Sierpinski en paso 1.275



Modelo Sierpinski en paso 1.533

La ejecución del modelo y unos cuantos giros alrededor de la estructura resultante nos darán mejor idea de lo comentado anteriormente, pero puede apreciarse en la imagen que la estructura de la pirámide se encuentra replicada, cada vez a menor escala, en las partes mas pequeñas de la representación.

Dado que nuestro metasimulador posiciona prismas hexagonales en posiciones enteras no podremos obtener una visión dentro del campo real pero la estructura, incluso resuelta en el campo de los enteros, deja a las claras la belleza del modelo generado.

La satisfacción por el modelo conseguido no hizo sino espolear las ganas de probar con algún otro fractal IFS un poco más complejo. Dado que Sierpinski solo utiliza una función afín de transformación, se buscó algún modelo que presentara un sistema de funciones variado. El siguiente modelo presenta un sistema de cuatro funciones de transformación, pero es su resultado lo que me animó a incluirlo. El resultado que obtendría sería mi tan ansiado helecho.

En la década de los 80, el investigador y emprendedor Michael F. Barnsley, interesado en la teoría de fractales y, más concretamente, en los sistemas de funciones iteradas y la compresión fractal de imágenes, propuso un algoritmo IFS con cuatro funciones de transformación sobre el plano que generaban lo que parecía un fabuloso *Asplenium adiantum-nigrum* de la familia de los helechos. En teoría, todo el esfuerzo para construir este modelo se debe única y exclusivamente al sistema de ecuaciones propuesto que es como sigue:

Sea x_i un valor de abcisa en el intervalo [-1,1]

Sea (x_0,y_0) el punto inicial en (0,0)

Para todo (x_i,y_i), i=1..n, hacer:

Se elije un valor de probabilidad p en el intervalo [0,1]

Si $0 \leq p \leq 0.01$, aplicar la función de transformación:

$$x_{n+1} = 0$$
$$y_{n+1} = 0.16 * y_n$$

Si $0.01 < p \leq 0.08$, aplicar la función de transformación:

$$x_{n+1} = 0.2 * x_n - 0.26 * y_n$$
$$y_{n+1} = 0.23 * x_n + 0.22 * y_n + 1.6$$

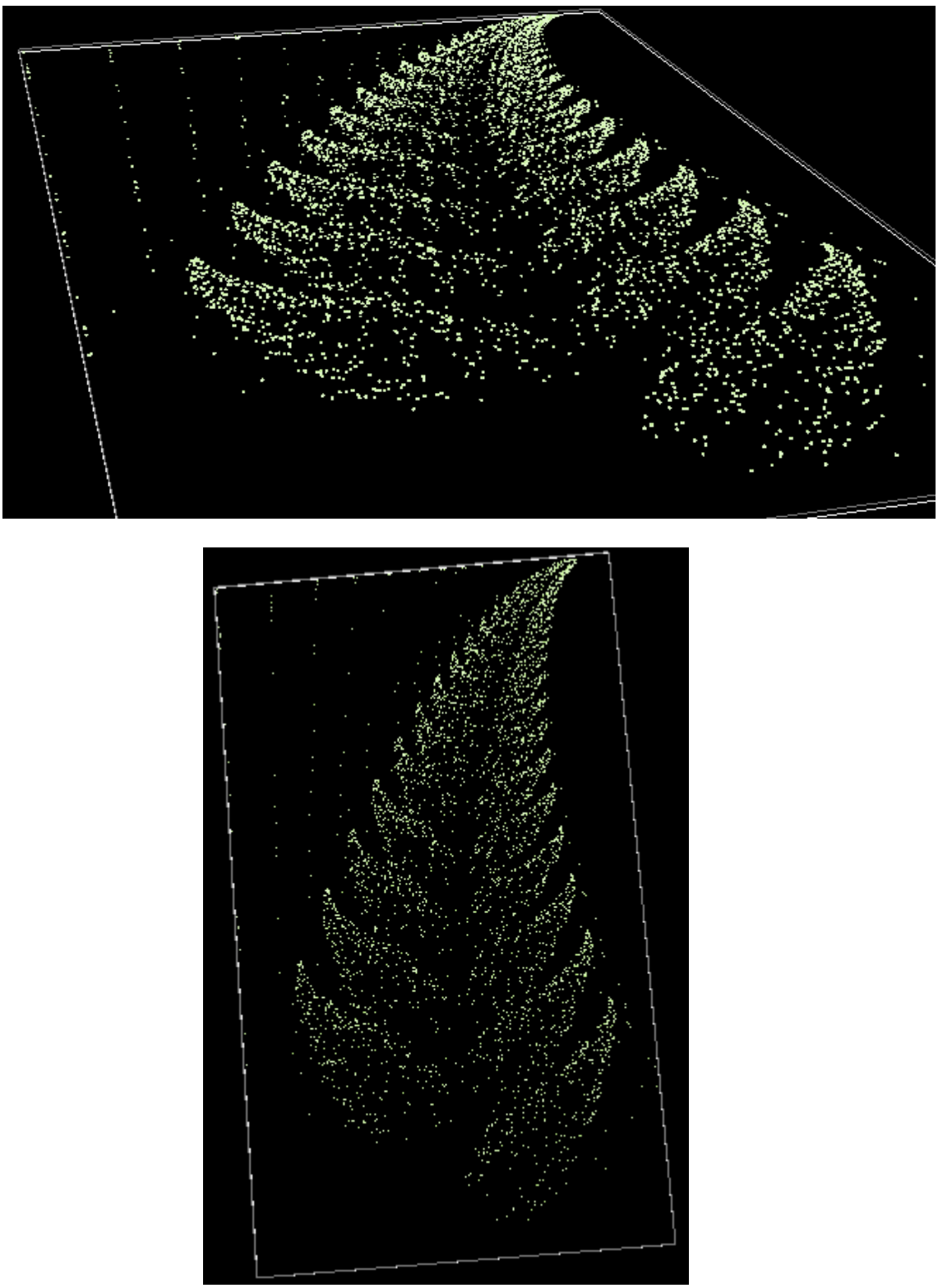
Si $0.08 < p \leq 0.15$, aplicar la función de transformación:

$$x_{n+1} = -0.15 * x_n + 0.28 * y_n$$
$$y_{n+1} = 0.26 * x_n + 0.24 * y_n + 0.44$$

Si $0.15 < p \leq 1$, aplicar la función de transformación:

$$x_{n+1} = 0.85 * x_n + 0.04 * y_n$$
$$y_{n+1} = -0.04 * x_n + 0.85 * y_n + 1.6$$

Esta serie de transformaciones aplicadas generan el helecho deseado durante tanto tiempo.



Modelo Barnsley en paso 14.880

Tras este helecho y como colofón de este capítulo no pude dejar pasar la creación de un modelo de fractal IFS con estructura de árbol.

Al fin y al cabo, gran parte de la memoria está repleta de ejemplos de la famosa manzana cayendo del árbol y me permití la licencia de anotar un fractal más a la colección de ejemplos.

Su método de obtención es muy similar al del helecho de Barnsley por lo que tan solo se describirán las funciones empleadas:

Sea x_i un valor de abcisa en el intervalo [-1.5,10.5]

Sea (x_0,y_0) el punto inicial en (0,0)

Para todo (x_i,y_i), i=1..n, hacer:

Se elije un valor de probabilidad p en el intervalo [0,1]

Si $0 \leq p < 0.2$, aplicar la función de transformación:

$$x_{n+1} = 0.19 * xn - 0.049 * yn + 0.44$$
$$y_{n+1} = 0.34 * xn + 0.44 * yn + 0.35$$

Si $0.2 \leq p < 0.4$, aplicar la función de transformación:

$$x_{n+1} = 0.46 * x_n + 0.41 * y_n + 0.25$$
$$y_{n+1} = -0.25 * x_n + 0.36 * y_n + 0.57$$

Si $0.4 \leq p < 0.6$, aplicar la función de transformación:

$$x_{n+1} = -0.06 * x_n - 0.07 * y_n + 0.59$$
$$y_{n+1} = 0.45 * x_n - 0.11 * y_n + 0.09$$

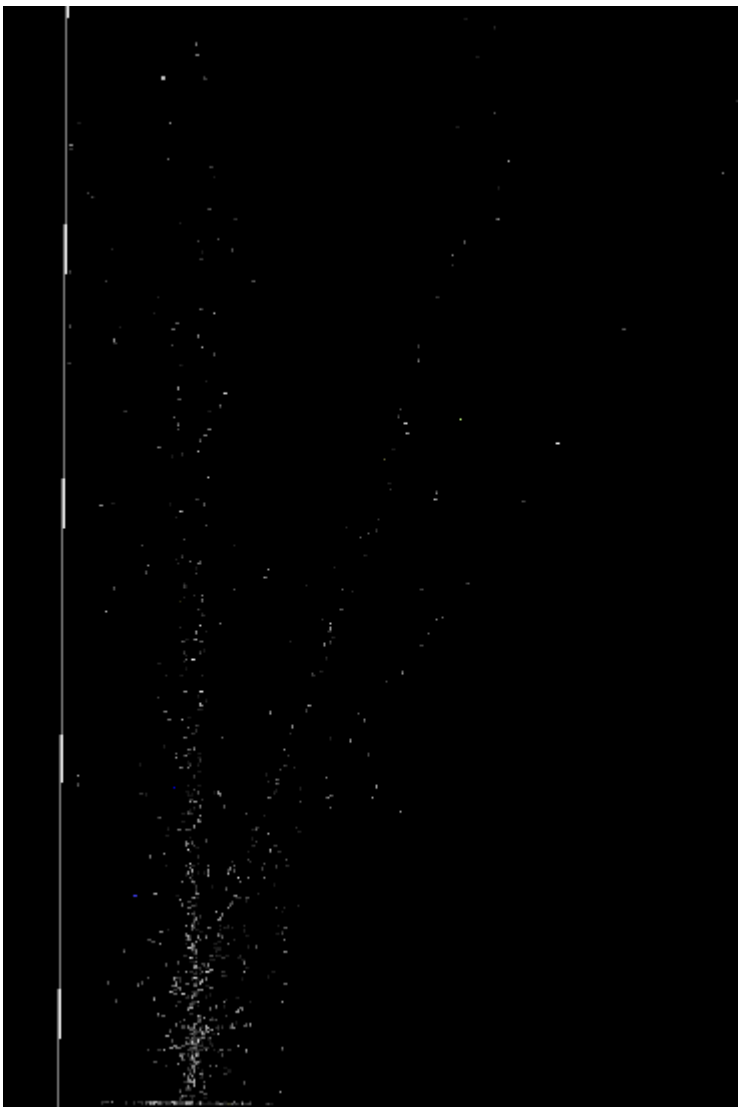
Si $0.6 \leq p < 0.8$, aplicar la función de transformación:

$$x_{n+1} = -0.03 * x_n + 0.07 * y_n + 0.49$$
$$y_{n+1} = -0.47 * x_n + 0.02 * y_n + 0.51$$

Si $0.8 \leq p < 1$, aplicar la función de transformación:

$$x_{n+1} = -0.64 * x_n + 0.86$$
$$y_{n+1} = 0.50 * y_n + 0.25$$

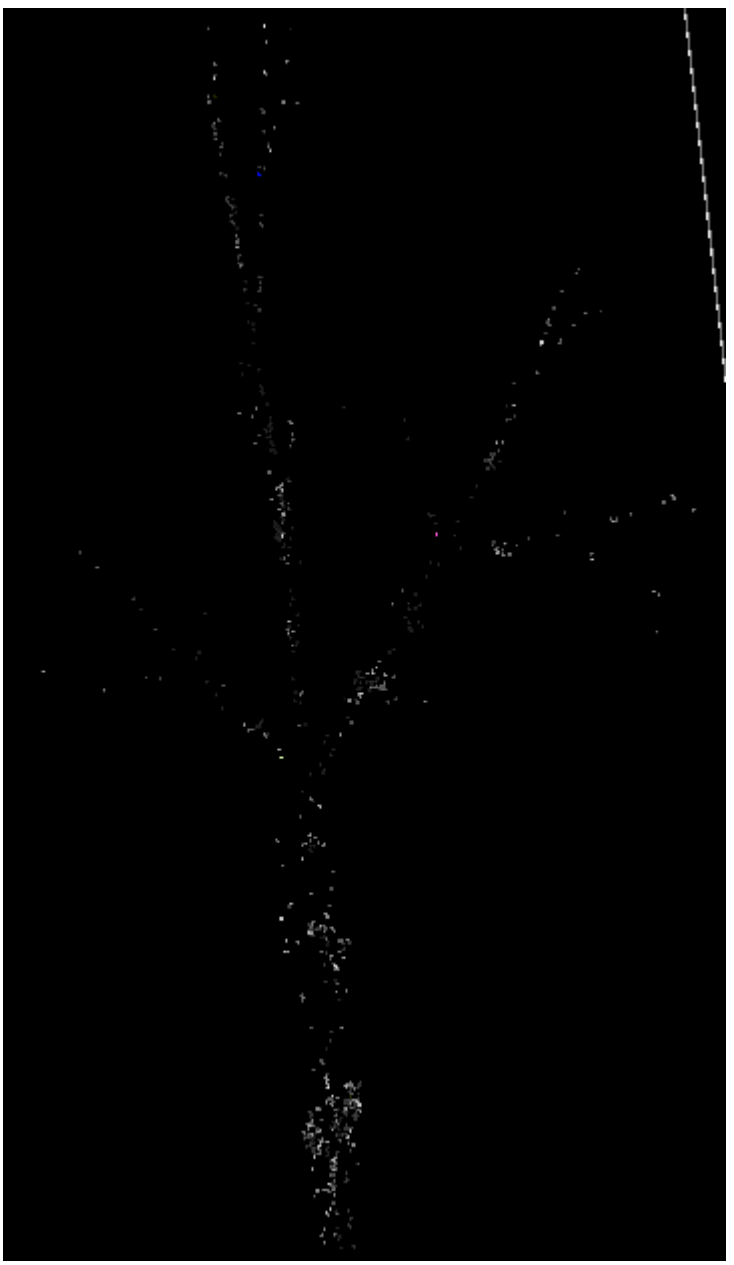
El ejemplo **Arbol_1.conf** escrito según el algoritmo anterior nos muestra, en su ejecución, una estructura con aspecto de dendrita neuronal o, bajo mi punto de vista, de árbol caduco en invierno.



Modelo Arbol_1 en paso 18.933

Los siguientes modelos de árboles no son sino variaciones del fractal anterior con distintos valores para las funciones de transformación.

Así el modelo **Arbol_2.conf** nos proporciona una estructura de tronco mas gruesa y, en general, mas centrada en el espacio de representación.



Modelo Arbol_2 en paso 7.115

El modelo **Arbol_3.conf** nos devuelve esta misma estructura, ahora si, con mayor aspecto de dendrita neuronal que de árbol, si bien no reconocemos una base típica y sus ramas presentan excesiva simetría en la representación, lo que no quita para que sea incorporado como cierre de los distintos ejemplos de este tipo de modelo que completan la serie presentada en este proyecto.

Modelo Arbol_3 en paso 2.264

14 MODELOS TIPO VI

Imágenes
Píxeles hexagonales

En anteriores apartados de la memoria se comentó que el simulador posee la capacidad de representar una imagen dada por su fichero de imagen en el archivo de configuración. La carga y representación en sí es un hecho bastante interesante bajo el punto de vista de la riqueza funcional del programa, pero se hace merecedor de un tipo de modelo cuando intentamos trabajar sobre la imagen cargada en la representación.

En este capítulo se describirán algunos modelos de tratamiento de imágenes que pongan de relieve, una vez más, la potencia subyacente del metasimulador extendido.

Como ha quedado claro en anteriores ejemplos, el metasimulador tiene limitaciones en cuanto a su velocidad de ejecución. Esta limitación es evidente cuando el número de elementos a representar crece de pocos centenares a miles de elementos.

En este tipo de modelos este problema se presentará casi de continuo en cualquiera de los ejemplos, ya que una mínima imagen de, por ejemplo, 300 por 300 píxeles supone el manejo dentro del espacio de representación de 90.000 elementos gráficos y matemáticos.

Si pensamos al respecto de las soluciones encontradas para los modelos del capítulo anterior, basadas en la activación de la propiedad de parada de la inmensa mayoría de los elementos representados al no ser necesario un cálculo sobre los mismos, veremos que en los modelos de este tipo no podremos utilizar esta misma solución ya que ahora si necesitaremos calcular distintas propiedades para cada ficha, principalmente las variables intrínsecas de color de cada elemento.

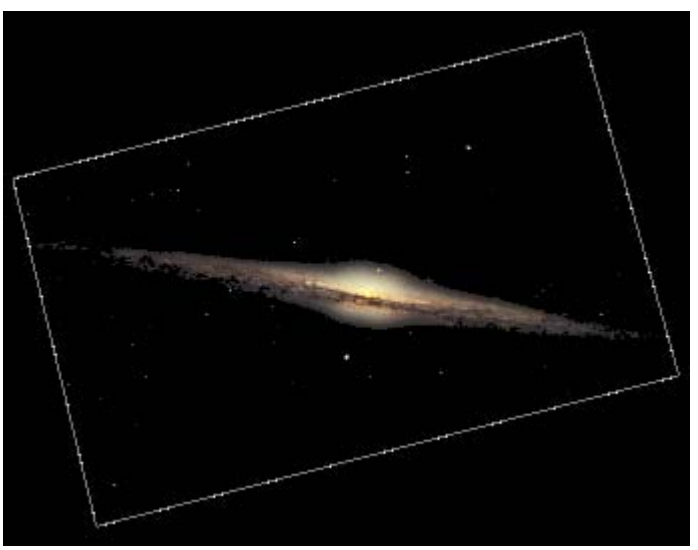
Esto supone que el modelo de ejemplo requeriría en principio el cálculo de 21 variables de color, tres colores RGB para el centro y las seis aristas de las bases del prisma, en cada una de las 90.000 fichas existentes. Estamos hablando de calcular 270.000 valores en cada paso de ejecución. Esto obligará a tomar ciertas decisiones de diseño, establecer algunas restricciones y realizar un número discreto de pasos de ejecución, como se podrá observar en los distintos ejemplos presentados en este capítulo, para poder llegar a un compromiso que nos permita representar imágenes basadas en elementos hexagonales y trabajar con estas.

La primera decisión, de diseño, guarda relación con las primeras pruebas de ejecución de gráficos de la aplicación inicial. La carga de una imagen de la galaxia Aguja o NGC 4565, de 672 por 471 píxeles supone la lectura y representación de 316.512 elementos dentro del espacio de simulación. Es casi obligatorio reducir el ancho del eje Z a tan solo dos niveles, lo que supone la existencia de 633.024 casillas que el sistema debe manejar en memoria. Eso es un volumen de memoria que casi impide la ejecución por falta de espacio para las estructuras de datos del simulador. Sin embargo, el uso de ejemplos basados en galaxias dio la pista para una mejora fundamental del simulador.

Observemos la imagen original de nuestra galaxia:

Galaxia Aguja, NGC 4565

y observemos la imagen representada por el simulador:



Modelo Img_NGC4565 en paso 0

como es evidente, parece que el simulador, conociendo las exigencias de memoria planteadas anteriormente, es capaz de representar la imagen de nuestra galaxia. Esto es posible gracias al color predominante de este tipo de imágenes.

Efectivamente, el color negro es el color predominante de la imagen y, además, es el color por defecto del espacio de representación del simulador. Tan solo deberemos realizar una ligera modificación en el código del simulador extendido para que no sean tenidos en cuenta aquellos elementos con color negro o cercano al negro.

De hecho, el parámetro de corte se sitúa en valores RGB en el rango [0,0,0]..[40,40,40] de forma que no ocupen lugar ni en memoria ni en los cálculos que cada simulación precise.

En el caso del modelo elegido estamos hablando de pasar de 316.512 elementos a tan solo 25.532 fichas para el modelo, lo que supone aproximadamente el 8% del total de elementos.

Es evidente la ganancia en espacio de memoria y tiempos de ejecución bajo esta restricción impuesta al sistema y sin la que sería casi imposible poder haber realizado los modelos que se describirán a continuación.

Utilizando el orden de creación de cada ejemplo, comenzaremos por el modelo llamado **Img_Media.conf** que intenta ejecutar un algoritmo sencillo de difuminado de la imagen dada.

Este modelo tiene unas instrucciones sencillas para obtener el resultado:

Creación de la Clase que alberga los elementos, píxeles, de la imagen:

[CLASE_DEF]

I

Declaración de las funciones para cada píxel de la imagen:

[CLASE_VAR]

[CLASE_FUNC]

I.C0r={Media(I.3>C0r)}
I.C0g={Media(I.3>C0g)}
I.C0b={Media(I.3>C0b)}
I.C1r=I.C0r
I.C1g=I.C0g
I.C1b=I.C0b
……
I.C6r=I.C0r
I.C6g=I.C0g
I.C6b=I.C0b

Declaración de cada píxel de imagen como elemento de la Clase:

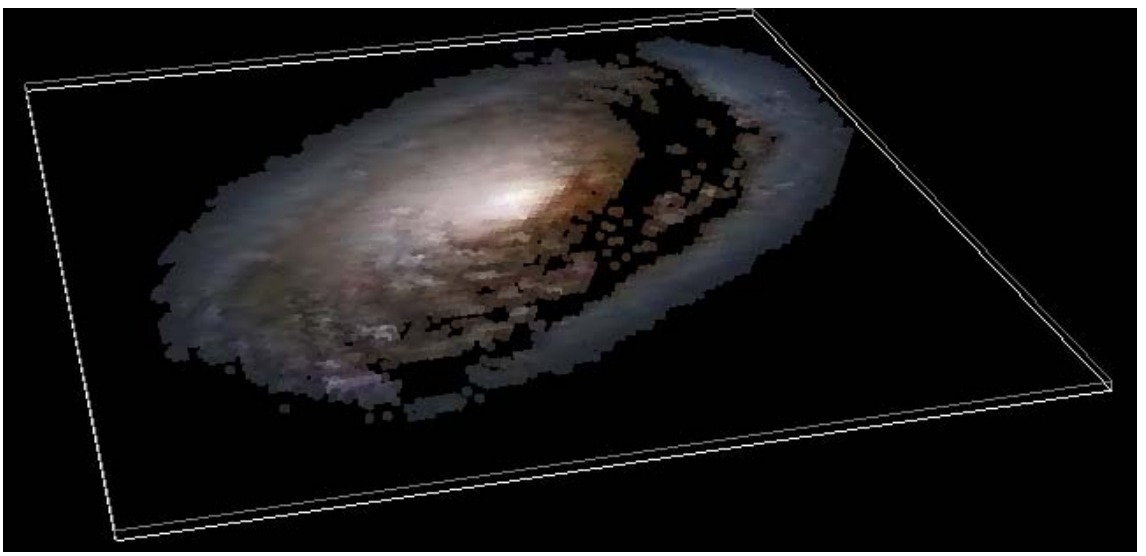
[FICHA_DEF]

IMG Imagen.jpg I 1 1

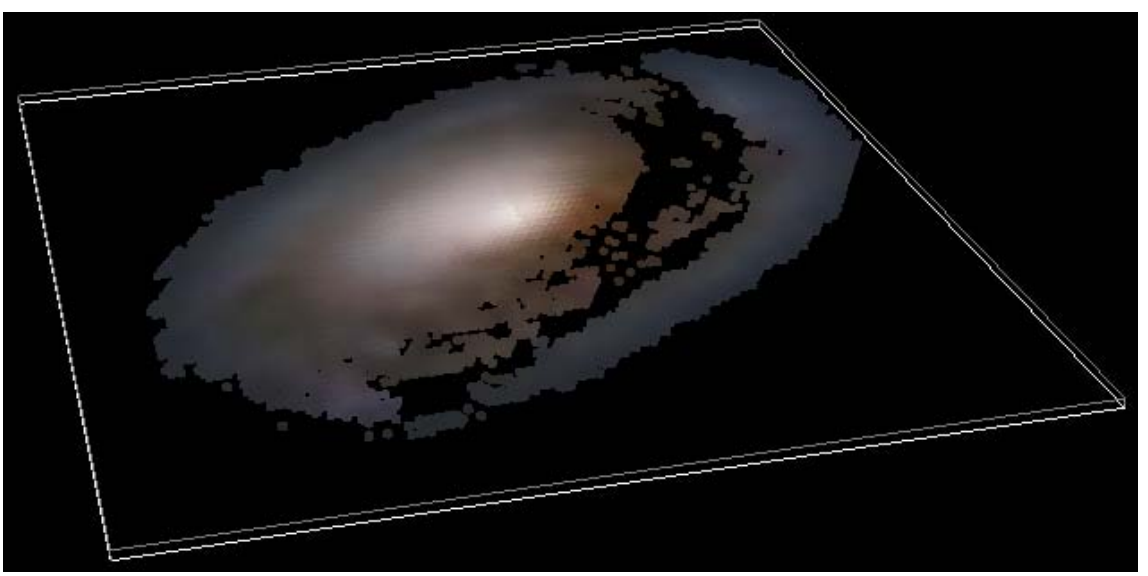
Dado que el metasimulador lee un punto de imagen y establece el color leído como color único para todo el prisma hexagonal, nos basta con calcular la media de cada componente RGB de color del punto central en un círculo de radio 3, asignarlo como el nuevo componente de color RGB para cada ficha y asignarlo incondicionalmente al resto de

variables de color I.Cir, I.Cig, I.Cib con i = 1..6 de forma que obtengamos en cada elemento un único color que es media de los colores de su entorno cercano.

Este sencillo cálculo hace que la imagen se difumine, tal como podemos apreciar en los primeros pasos de ejecución:

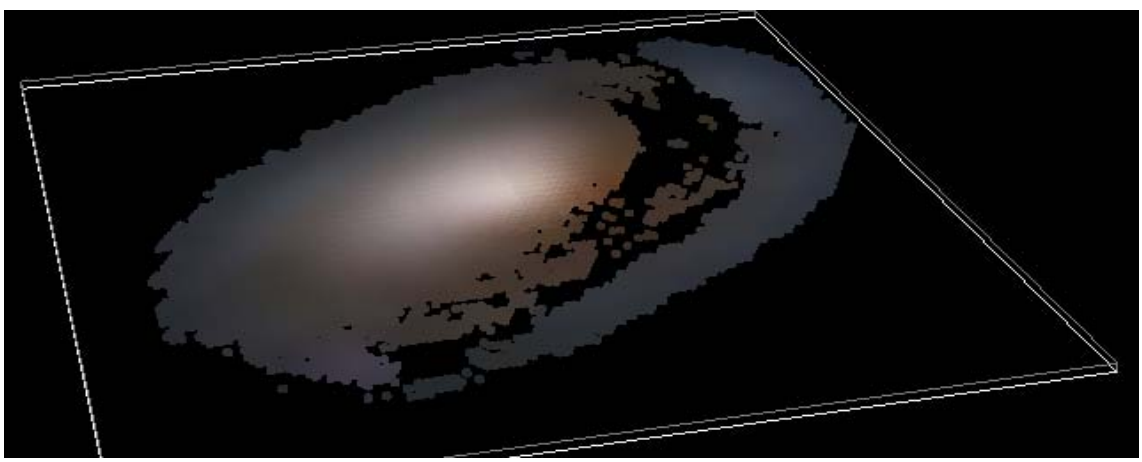


Modelo Img_Media en paso 0



Modelo Img_Media en paso 1

Modelo Img_Media en paso 2



Modelo Img_Media en paso 3

Modelo Img_Media en paso 4

Como puede observarse en la secuencia de imágenes de los primeros pasos de ejecución, no existen grandes diferencias a partir del primer paso ejecutado, ya que los valores locales por elemento tienen casi desde el principio un valor idéntico a la media obtenida del primer paso.

Se podría decir que este modelo es de un único paso de ejecución y casi podría añadirse que debe serlo, ya que la ejecución de un solo paso lleva más de tres minutos de ejecución en un potente ordenador.

Esto es debido, tal como ya se comentó al comienzo de este capítulo, al considerable número de elementos existentes que hacen que un solo paso de ejecución deba calcular más de quince funciones por elemento, entre un conjunto de varios miles de elementos. Incluso con el filtro de elementos negros comentado con anterioridad se hace casi imposible mostrarlo de otra forma que no sea mediante secuencia de imágenes.

Podemos suponer que sería de esta ejecución si tuviera que calcular, además de los ya existentes, todos los elementos negros existentes en la imagen original.

El modelo realizado tras el que se acaba de describir es algo más ligero en ejecución. El nombre del mismo es **Img_DelBlack.conf** y, como su nombre indica, pretende extender el filtro existente en el sistema a valores superiores de RGB, pudiendo llegar a hacer desaparecer la imagen. Puede decirse que realiza un control de contraste de la imagen dada.

Su archivo de configuración es como sigue:

Coeficientes para filtro de valores RGB a eliminar:

```
[COEF_VAR]
COEF0 0 255 70
COEF1 0 255 70
COEF2 0 255 70
```

Creación de la Clase que alberga los elementos, píxeles, de la imagen:

```
[CLASE_DEF]
I
```

Condición de eliminación de los píxeles con RGB inferior al dado:

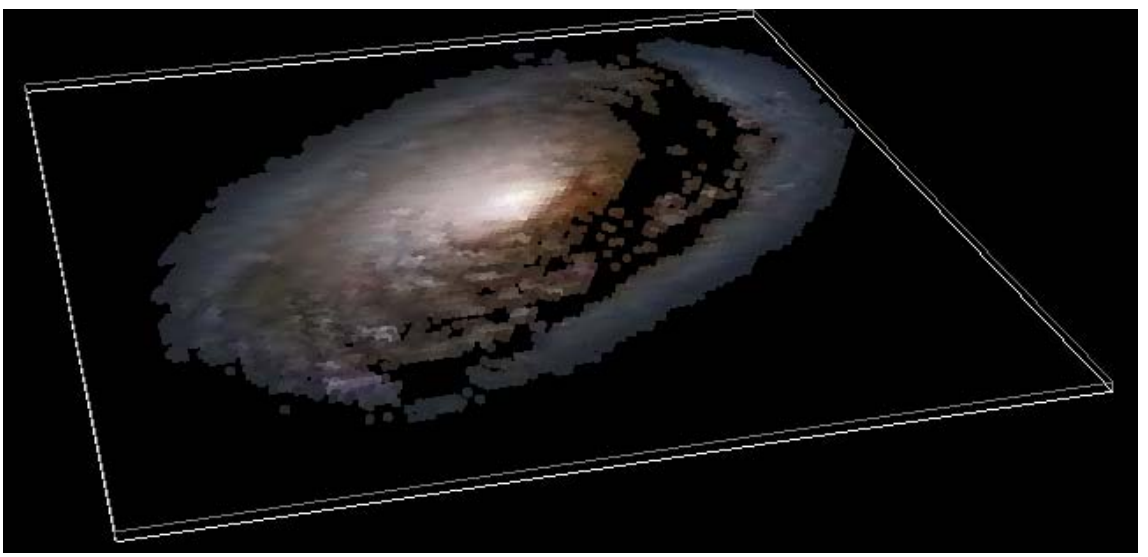
```
[CLASE_SUPR]
(I.C0r<COEF0&&I.C0g<COEF1&&I.C0b<COEF2)
```

Declaración de cada píxel de imagen como elemento de la Clase:

```
[FICHA_DEF]
IMG Imagen.jpg I 1 1
```

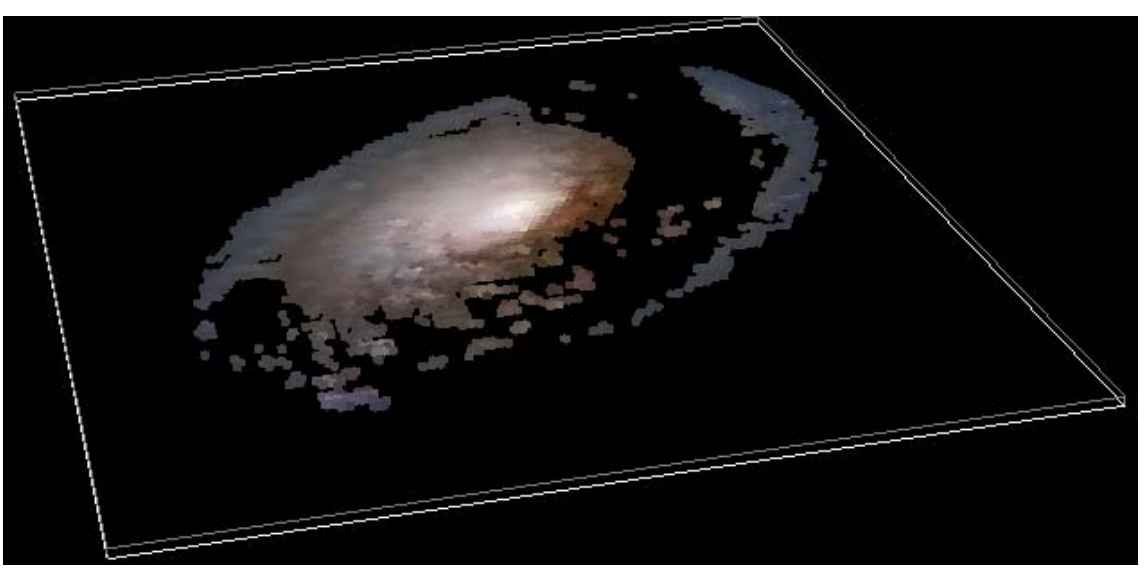
Tal como puede apreciarse, definimos un valor RGB de corte que elimina de la imagen todos los elementos con color inferior a dicho valor, de forma que tan solo perduran aquellos con valor superior.

Partimos, como en el ejemplo anterior, de la misma imagen original:



Modelo Img_DelBlack en paso 0

y dejamos ejecutar el primer paso que sitúa el corte en un valor de color dado por los valores por defecto de los coeficientes de RGB = [0,0,0]..[70,70,70] con lo que se obtiene la siguiente representación:



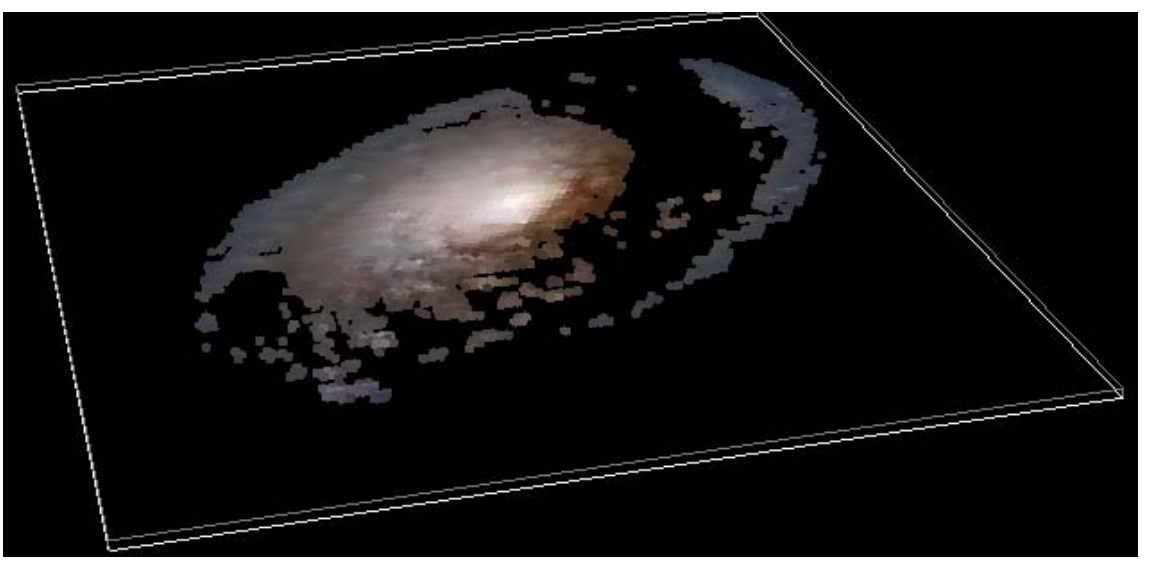
Modelo Img_DelBlack en paso 1

Podemos observar que aquellos puntos de color RGB inferior al del corte han sido eliminados de la representación. Si aumentamos el filtro de corte hacia un valor superior, digamos [141,141,141] el modelo resultante queda representado por la siguiente imagen:

Modelo Img_DelBlack en paso 2

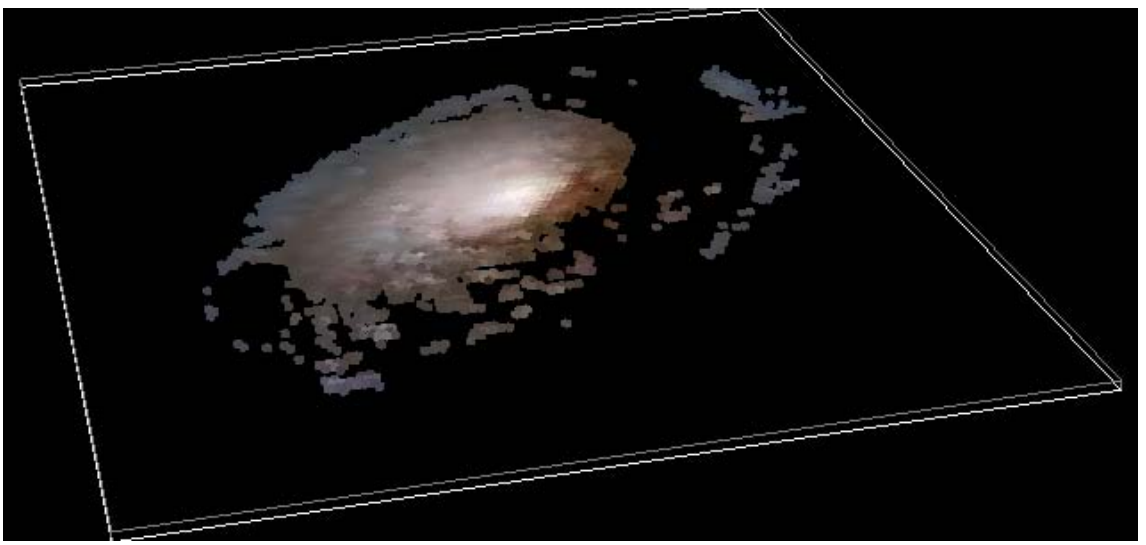
Nuestra imagen, originalmente bastante oscura, ha quedado casi eliminada de la representación, en la que solo subsisten aquellos elementos mas brillantes. Parece evidente que obtendremos una simulación sin fichas si elevamos el filtro de corte unos cuantos enteros de forma que exijamos que solo los colores cercanos al blanco puedan mantenerse en el modelo ejecutado. Dado que esto es obvio prescindiremos de la imagen correspondiente y buscaremos combinaciones de los valores RGB del filtro de corte para la imagen.

Así, si ejecutamos el modelo variando tan solo el valor del componente G de color, verde puro, de forma que exigiéramos la no existencia de este color básico el modelo resultante sería muy similar al que ya obtuvimos en el primer paso, como puede observarse en la imagen:



Modelo Img_DelBlack (G=255) en paso 7

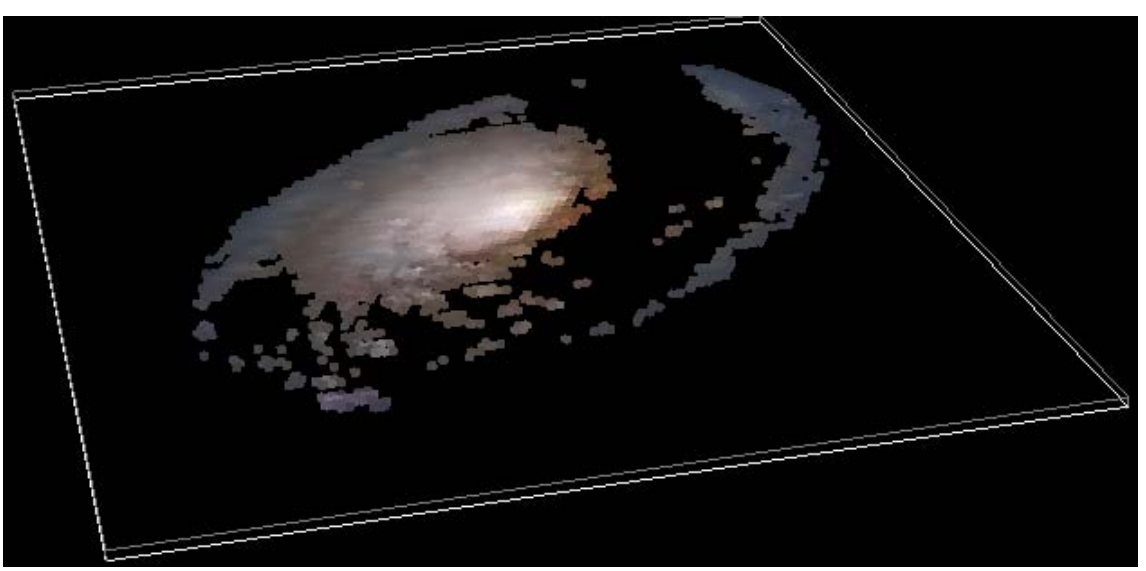
Esto no hace sino indicarnos que la imagen casi no dispone de colores combinación de la gama de verdes, con lo que no se altera considerablemente su resultado. Si realizamos un filtro sobre el componente azul, la representación queda tal como se observa en la imagen:



Modelo Img_DelBlack (B=255) en paso 2

Ahora, los brazos externos de la nebulosa, de alto componente azul, son los que desaparecen, manteniéndose los colores claros y los marrones.

Probemos a ejecutar el modelo con el filtro para valores de rojo puro. El resultante de la ejecución queda como la siguiente imagen:

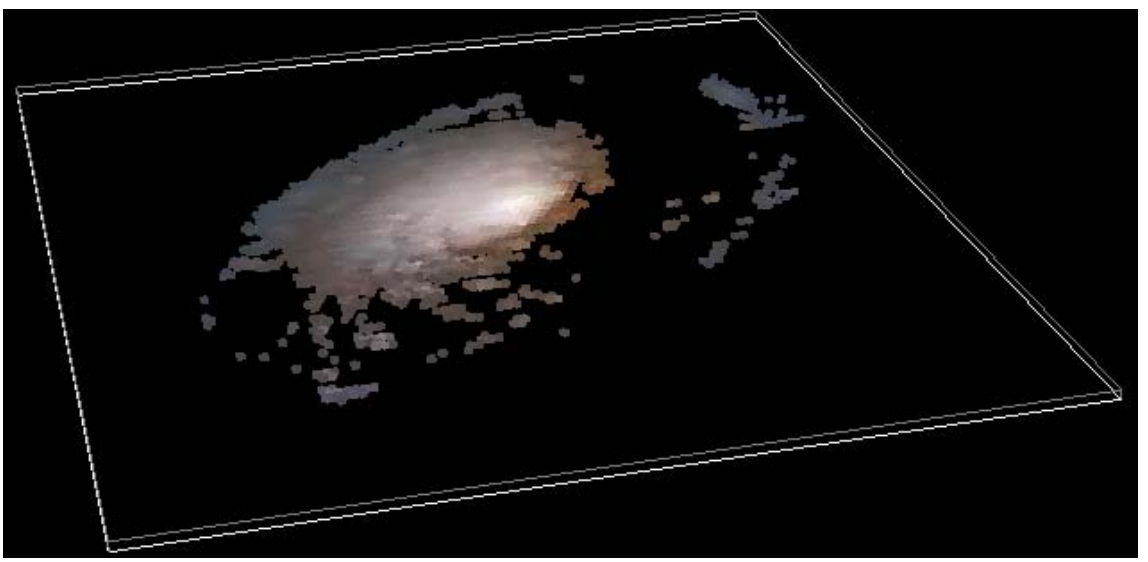


Modelo Img_DelBlack (R=255) en paso 5

La ejecución pone de relieve que la imagen tiene un alto componente de colores combinación de dos básicos, lo que hace que ninguno de los filtros de uno de los colores RGB arroje grandes diferencias.

Para comprobar la veracidad de la reflexión anterior observemos la imagen original y probemos a exigir una barrera para valores con fuerte mezcla de los colores básicos rojo y azul.

El resultado de aplicar un filtro de este estilo es la siguiente imagen:



Modelo Img_DelBlack (R=255,B=255) en paso 7

lo que nos indica que, aún eliminados algunos puntos mas con respecto al filtro anterior, nuestra imagen tiene dependencia de colores de combinación sobre los tres colores básicos.

Por ello, la primera de las ejecuciones es similar a esta y hubiera sido suficiente para este tipo de imágenes.

El siguiente modelo no es sino una mejora del anterior. Con el nombre **Img_UpdBlack.conf**, esta simulación elimina los elementos con valores RGB de color que sean menores que el filtro RGB dado pero no pierde estos elementos sino que los pasa a color negro puro, almacenando los valores de partida, tal como puede observarse en su fichero de configuración con las instrucciones siguientes:

Declaración de las variables de color original y de elemento activo:

[CLASE_VAR]
I.red=-1
I.blue=-1
I.green=-1
I.active=1

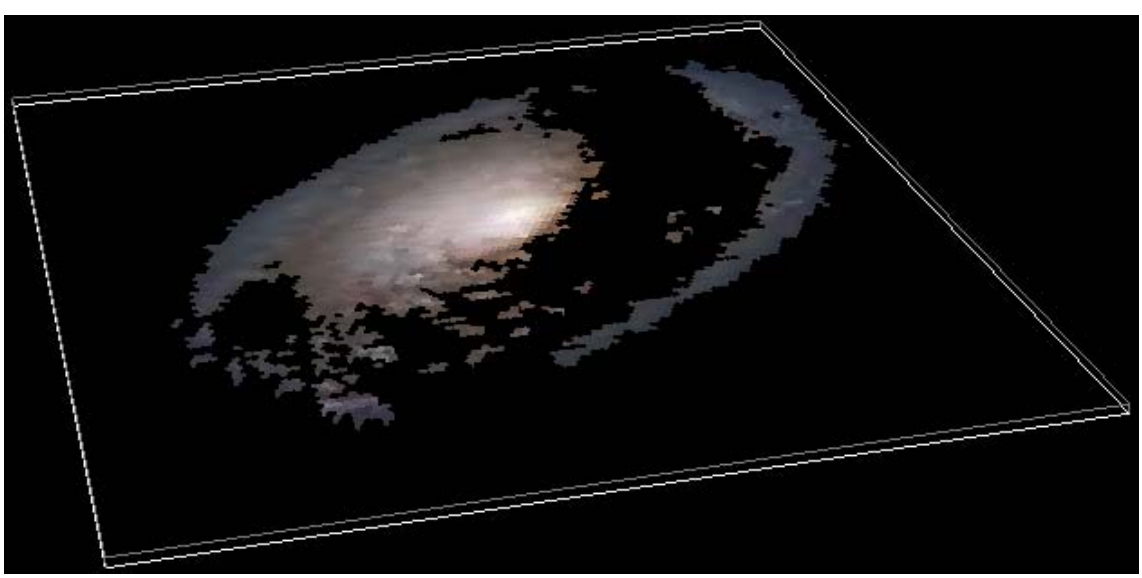
Declaración de las Funciones de almacenado de color y eliminación lógica:

```
[CLASE_FUNC]
I.red=if(I.red==-1,I.C0r,I.red)
I.green=if(I.green==-1,I.C0g,I.green)
I.blue=if(I.blue==-1,I.C0b,I.blue)
I.active=if(I.red<COEF0&&I.green<COEF1&&I.blue<COEF2,0,1)
I.C0r=if(I.active==1,I.red,0)
I.C0g=if(I.active==1,I.green,0)
I.C0b=if(I.active==1,I.blue,0)
I.C1r=I.C0r
I.C1g=I.C0g
I.C1b=I.C0b
……
I.C6r=I.C0r
I.C6g=I.C0g
I.C6b=I.C0b
```

Tal como nos muestra el listado anterior, si un elemento no supera el filtro RGB, se almacenan sus componentes de color RGB originales y se pasan a color negro puro en la ejecución.

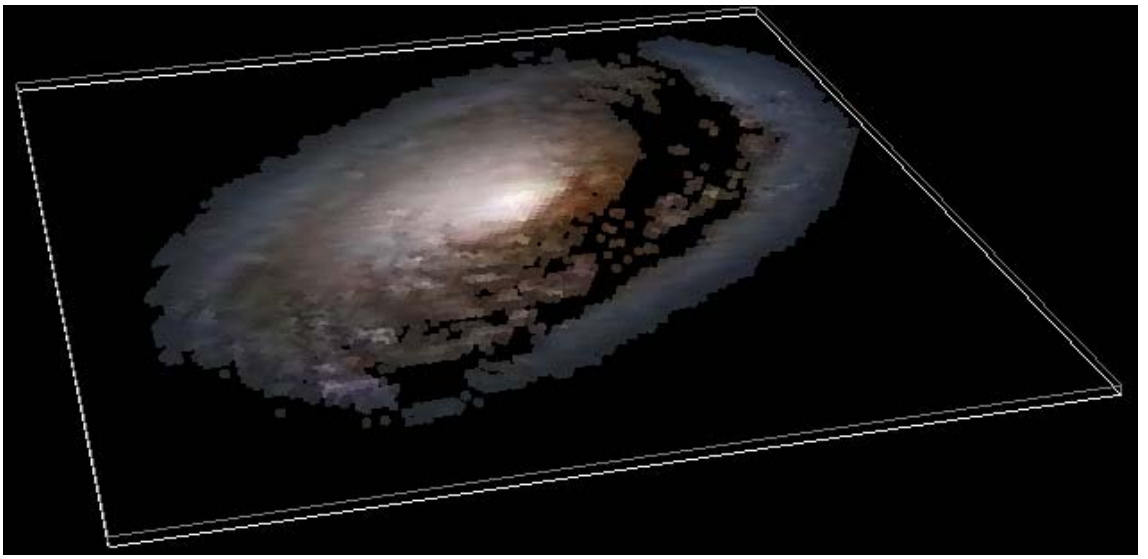
Si el filtro RGB es superado, entonces se mantiene el elemento original o, si había sido convertido a negro puro, se devuelve a su color de partida.

La ejecución de este modelo es mostrada desde un paso de ejecución cualquiera, en el que se han variado arbitrariamente los valores RGB del filtro, como puede observarse en la siguiente imagen:



Modelo Img_UpdBlack (R=192,G=114,B=63) en paso 1

Basta con bajar el filtro para que vuelva a recuperar los elementos perdidos, tal como puede observarse en la siguiente imagen:



Modelo Img_UpdBlack (R=39,G=39,B=39) en paso 2

de forma que el sistema vuelve a presentarnos la imagen casi original.

El siguiente modelo, de nombre **Img_Niveles.conf** supone un ejemplo interesante en cuanto a la separación de colores de una imagen.

Su algoritmo se describe mediante las siguientes instrucciones:

Coeficiente de separación de colores por altura:

[COEF_VAR]

COEF0 0 51 25

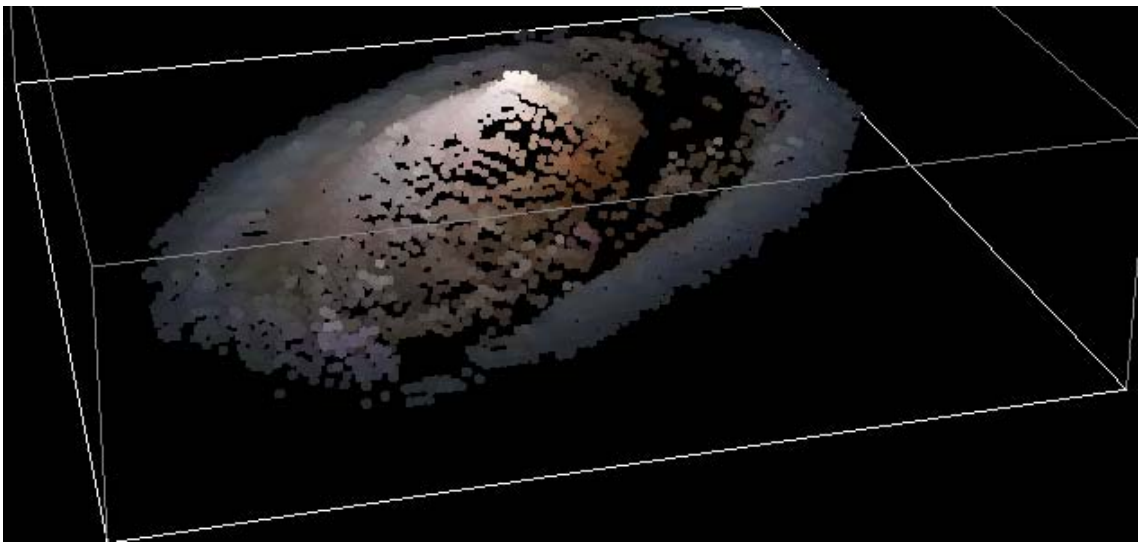
Función de colocación en plano Z según valor de color:

[CLASE_FUNC]

I.Z=(I.C0r+I.C0g+I.C0b)/COEF0

Esta sencilla función se encarga de cuantificar el rango de valores de cualquier color RGB, que pertenece al rango [0..255] de los números naturales, en 50 grupos de color, de forma que cada elemento se situará en una posición en el plano Z equivalente a uno de estos grupos.

Los valores 0 y 51 nos devolverán a la imagen de partida en ambos casos. Si ejecutamos el modelo con el coeficiente de grupo en su valor por defecto de 25, obtendremos la siguiente imagen:



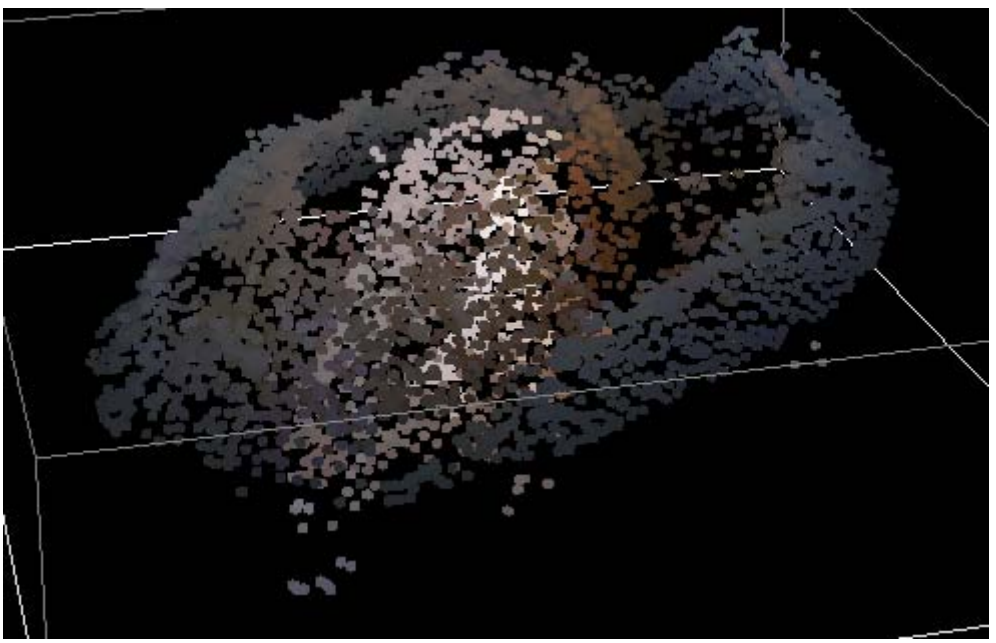
Modelo Img_Niveles (COEF=25) en paso 1

Puede observarse que los colores más claros, los más alejados de los azules y marrones oscuros, tienden a elevarse en el plano Z. Si movemos el coeficiente a valores próximos a 1 obtendremos resultados tridimensionales significativos como muestra la siguiente imagen:

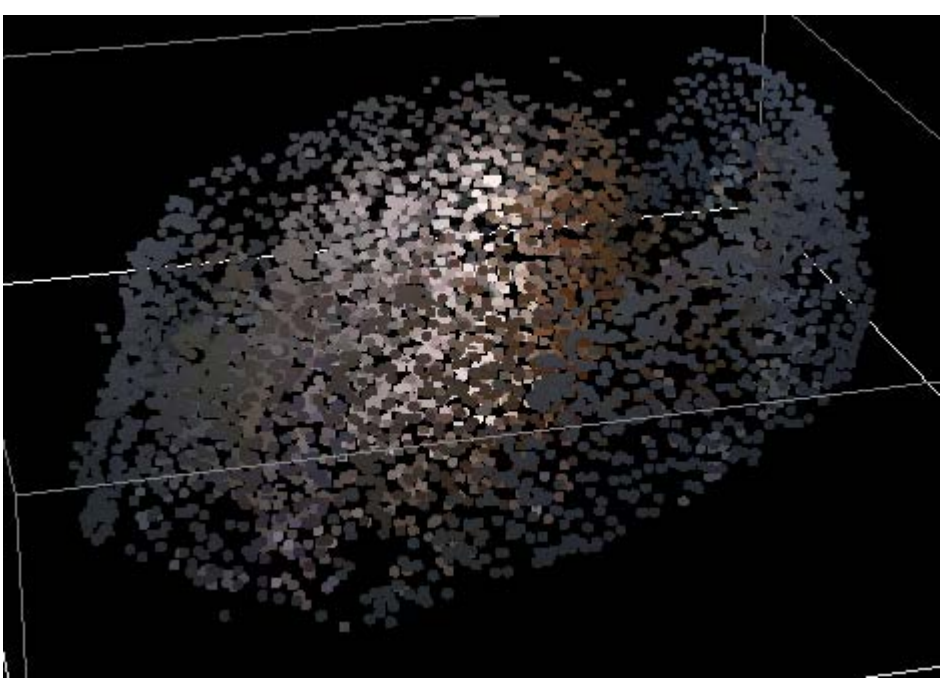
Modelo Img_Niveles (COEF=12) en paso 2

Puede observarse que los píxeles de colores cercanos al blanco quedan relegados a planos Z cercanos a cero mientras que los marrones mas claros ascienden niveles del eje Z de forma normal.

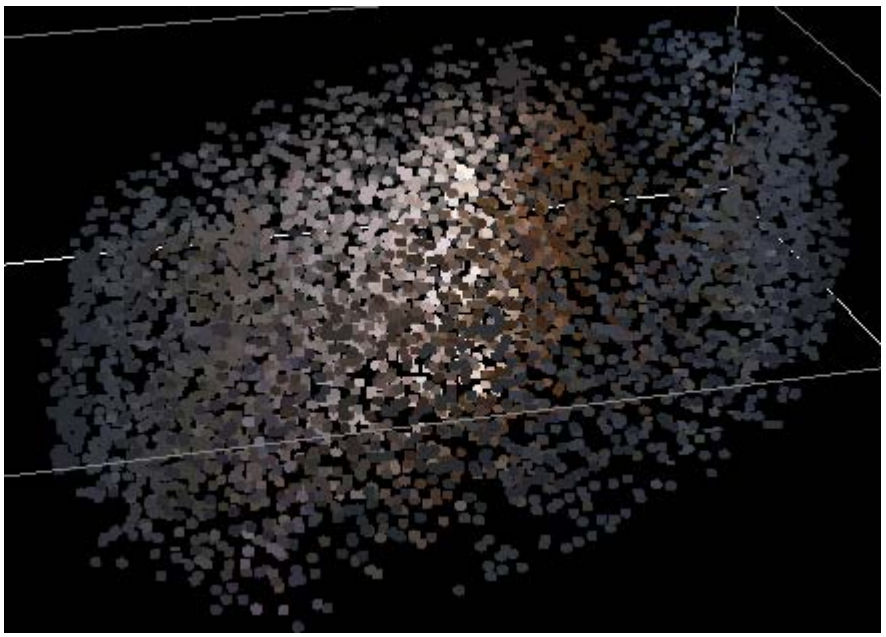
Si acercamos más el coeficiente a valores cercanos al 1 el efecto tridimensional de separación de puntos queda patente, como se muestra en la siguiente imagen:



Modelo Img_Niveles (COEF=5) en paso 3



Modelo Img_Niveles (COEF=2) en paso 4



Modelo Img_Niveles (COEF=1) en paso 5

Volviendo a la imagen original, como ya se ha indicado, para un valor de coeficiente igual a 0 o al máximo de los grupos creados, en este caso 51, lo cual incide sobre las características de la imagen, en la que ciertas gamas de color están abundantemente presentes y reduzcan el interés a los grupos 1 al 10 eminentemente.

Los restantes modelos de este tipo se encuentran en los límites de la computación para el sistema.

Debido a la cantidad ingente de elementos, a las numerosas operaciones basadas en funciones extendidas, que al no ser propietarias del compilador ejecutan varios órdenes de veces mas lentas, al uso de imágenes de un tamaño excesivo para la memoria o a los numerosos pasos de ejecución necesarios para obtener un resultado concreto, hacen que tan solo podamos describirlos mínimamente como frontera para este tipo de modelos.

El siguiente modelo, primero de los ejemplos límite de este tipo, con nombre **Img_Mix.conf** dispone dos imágenes distintas en dos niveles diferentes del eje Z y obtiene, para un valor de Z superior, la imagen resultante de la mezcla de color de los elementos coincidentes para las posiciones XY en cada uno de los dos planos anteriores, que se utilizan a modo de capas de una imagen.

El consumo de recursos y tiempo necsarios para ejecutar el modelo obliga a restringir el proceso de mezcla de forma que tan solo una pequeña área rectangular será objeto de la simulación, pudiendo deducirse perfectamente de la misma, como sería el resultado en caso de ampliarlo al total del espacio de representación.

Esta amplicación llevaría a un consumo de decenas de horas y, probablemente, a la parada del programa por falta de memoria.

La descripción del modelo es como sigue:

Declaración y uso de un Coeficiente de mezcla, inicialmente media de colores:

```
[COEF_VAR]
COEF0 1 30 20
[GLOBAL_VAR]
level=COEF0
[GLOBAL_FUNC]
level=COEF0/10
```

Declaración de las tres clases intervinientes, dos de capa y una de resultado:

```
[CLASE_DEF]
I
J
M
```

Variables de actividad de los elementos de capa y resultado y existencia de este:

```
[CLASE_VAR]
I.active=0
J.active=0
M.active=0
M.existI=0
M.existJ=0
```

Funciones de obtención de resultado como mezcla de elementos de capa:

```
[CLASE_FUNC]
M.active=if({Indice()}==0,1,0)
M.X=if(M.X<=180,M.X+1,120)
M.Y=if(M.X==180,M.Y+1,M.Y)
M.existI=if({EsHueco({M.X},{M.Y},1)}==0,1,0)
M.existJ=if({EsHueco({M.X},{M.Y},5)}==0,1,0)

M.C0r=if(M.existI==1,if(M.existJ==1,({Ficha({Indice({M.X},{M.Y},1)},C0r)}
+{Ficha({Indice({M.X},{M.Y},5)},C0r)})/level,{Ficha({Indice({M.X},{M.Y},1
)},C0r)}),if(M.existJ==1,{Ficha({Indice({M.X},{M.Y},5)},C0r)},0))

M.C0g=if(M.existI==1,if(M.existJ==1,({Ficha({Indice({M.X},{M.Y},1)},C0g)}
+{Ficha({Indice({M.X},{M.Y},5)},C0g)})/level,{Ficha({Indice({M.X},{M.Y},1
)},C0g)}),if(M.existJ==1,{Ficha({Indice({M.X},{M.Y},5)},C0g)},0))

M.C0b=if(M.existI==1,if(M.existJ==1,({Ficha({Indice({M.X},{M.Y},1)},C0b)
}+{Ficha({Indice({M.X},{M.Y},5)},C0b)})/level,{Ficha({Indice({M.X},{M.Y},
1)},C0b)}),if(M.existJ==1,{Ficha({Indice({M.X},{M.Y},5)},C0b)},0))
```

```
M.C1r=M.C0r
M.C1g=M.C0g
M.C1b=M.C0b
……
M.C6r=M.C0r
M.C6g=M.C0g
M.C6b=M.C0b
```

Nuevo elemento del resultado según mezcla para el elemento activo:

```
[CLASE_INS]

(M.active==1) M.X=Fpid.X M.Y=Fpid.Y M.Z=13 M.R=1 M.C0r=Fpid.C0r
M.C0g=Fpid.C0g M.C0b=Fpid.C0b M.C1r=Fpid.C0r M.C1g=Fpid.C0g
M.C1b=Fpid.C0b M.C2r=Fpid.C0r M.C2g=Fpid.C0g M.C2b=Fpid.C0b
M.C3r=Fpid.C0r M.C3g=Fpid.C0g M.C3b=Fpid.C0b M.C4r=Fpid.C0r
M.C4g=Fpid.C0g M.C4b=Fpid.C0b M.C5r=Fpid.C0r M.C5g=Fpid.C0g
M.C5b=Fpid.C0b M.C6r=Fpid.C0r M.C6g=Fpid.C0g M.C6b=Fpid.C0b
M.active=0 M.existI=0 M.existJ=0
```

Definición del elemento resultado y de los elementos de capa:

```
[FICHA_DEF]
0 120 120 12 1 M
IMG Galaxia2.jpg I 1 1
IMG Galaxia3.jpg J 5 1

[FICHA_COLOR]
0 1 2 255 255 255 255 255 255
```

Parada inicial de todos los elementos de capa y resultado menos el activo:

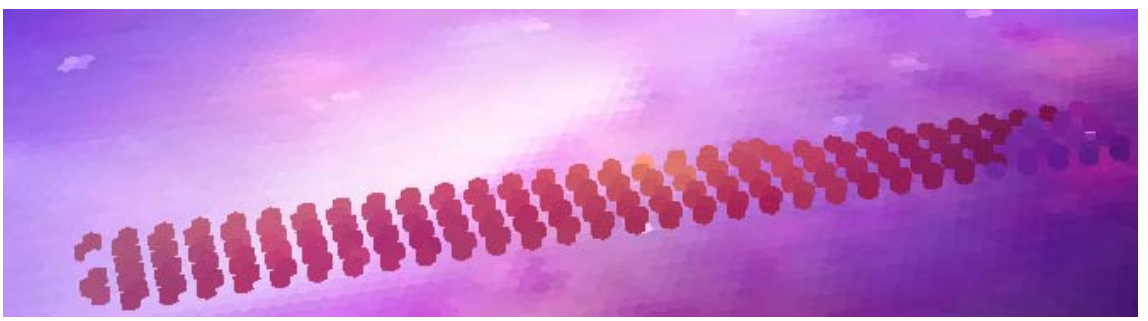
```
[FICHA_VAR]
[FICHA_FUNC]
[[CUARTA_PARTE]]

[CLASE_PARADA]
(I.active==0)
(J.active==0)
(M.active==0&&{Indice()}>0)
```

El algoritmo realiza una carga de elementos para las capas 1 y 2, como elementos de las clases I y J para cada una de las imágenes de capa e incorpora un único elemento de clase M, que realizará la mezcla de colores a presentar en el plano de resultados a medida que avance por el rango acotado para el modelo, localizado en un cuadrado de vértices [120,120]..[180,180] para su diagonal en el plano XY del resultado.

El primer paso de ejecución se encargará de parar todos los elementos correspondientes a las imágenes de partida para evitar que el sistema pase a calcular valores para estos y así ejecutar este modelo de forma más eficiente.

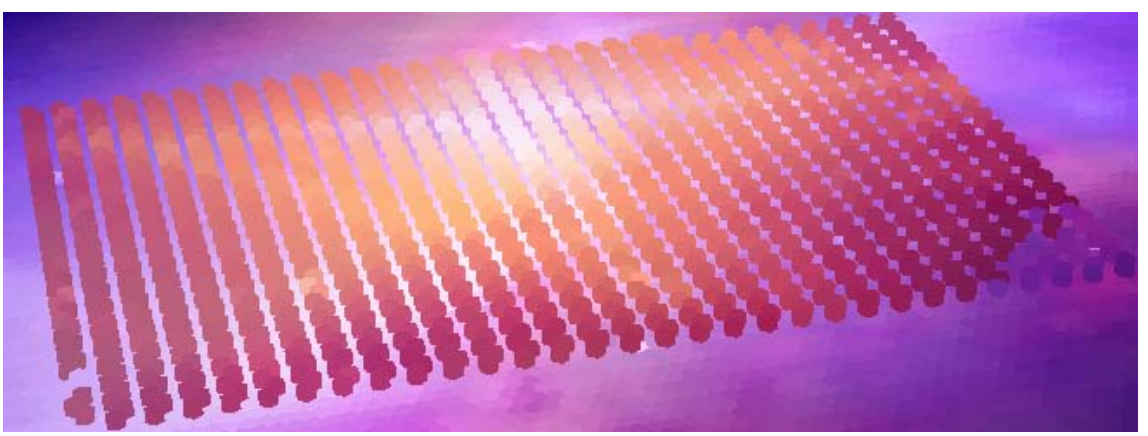
A partir de ahí, nuestro elemento de mezcla empieza a recorrer el rango dado y calcula la mezcla de color de los elementos en posiciones (X,Y,1) y (X,Y,5), es decir el punto con abcisa y ordenada iguales a las del elemento de mezcla pero con ejes Z correspondientes a cada plano de las imágenes que hacen de capas de la imagen a obtener.

En cada paso se inserta un nuevo elemento basado en su parent, que no es sino el elemento de la clase M activo en cada momento y desactiva el mismo, quedando este nuevo elemento como el encargado de la mezcla en el siguiente paso de ejecución. De esta forma obtenemos una imagen resultado que se va construyendo a medida que los pasos de ejecución se suceden. La ejecución, localizada en una región X_0Y_0-X_fY_f para aminorar el coste excesivo de la simulación, nos presenta en una rejilla no compacta pero suficiente el resultado de la mezcla de los elementos de color de ambos planos, como se muestra en la imagen:



Modelo Img_Mix en paso 474

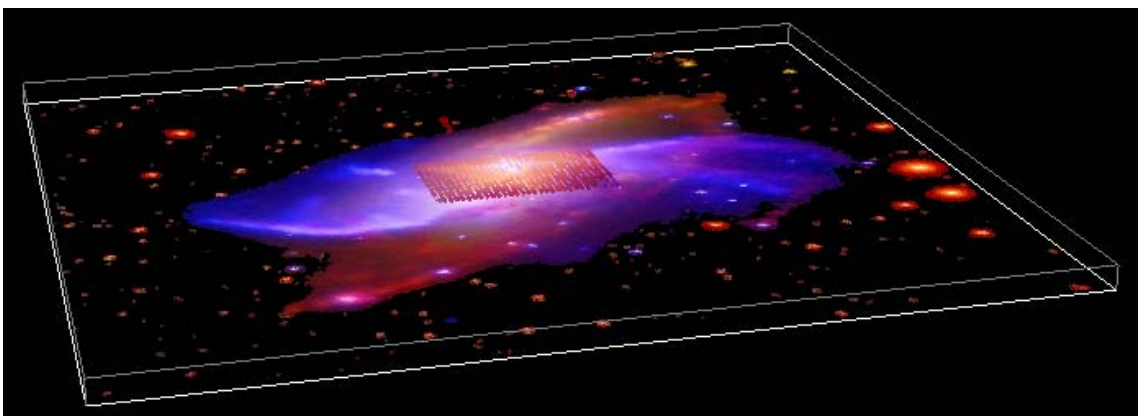
en la que se observan las distintas variaciones de color en función de cada par de elementos de partida. La ejecución de los siguientes pasos, completa la región resultado como se observa en la siguiente imagen:



Modelo Img_Mix en paso 2.649

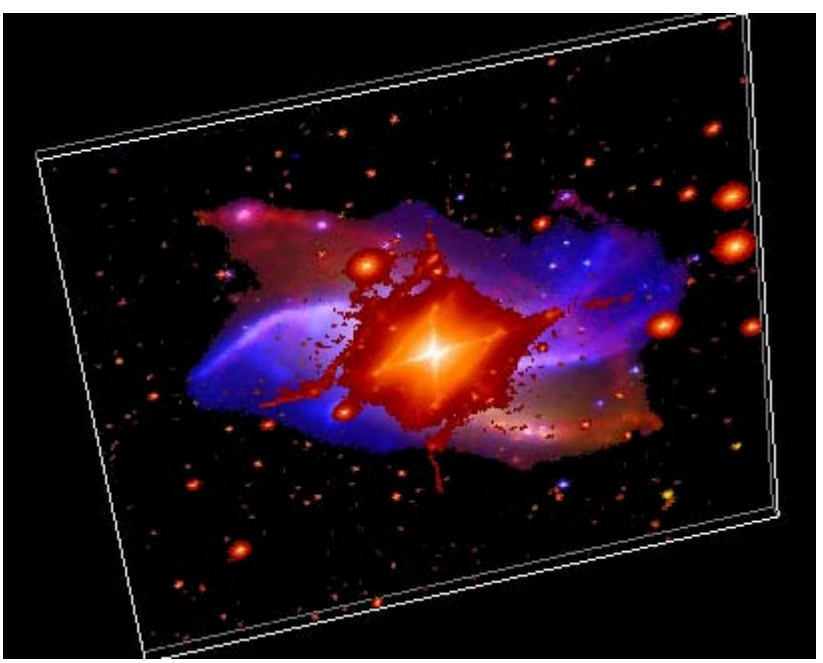
A partir de este paso, en la máquina donde se ha realizado este proyecto, el sistema completo se queda sin espacio de memoria para seguir procesando, lo cual nos da una idea de lo cercanos que estamos a los límites de ejecución de una máquina convencional.

Tal como se puede comprobar, la imagen resultante es una combinación de la imagen de la galaxia 2 que se muestra claramente debajo del resultado, tal como se aprecia en la siguiente imagen:



Modelo Img_Mix (Vista superior) en paso 2.649

y de la imagen situada en el plano Z inferior que puede verse si giramos el eje Z del modelo y miramos desde abajo al mismo, tal como se muestra en la siguiente imagen:



Modelo Img_Mix (Vista inferior) en paso 2.649

Se comprueba que los colores blancos, amarillos y rojos de la imagen inferior prevalecen en la imagen obtenida frente a los azules y rojos claros de la imagen superior.

El modelo siguiente es el último de los propuestos que han sido llevados a cabo, es decir, el último de aquellos ejemplos que suponen una variación cualitativa que apoye el supuesto planteado en este proyecto, al respecto de la potencia del simulador diseñado, y que pueden ser llevados a cabo dentro de los cercanos límites de computabilidad del sistema en una máquina comercial.

El modelo, de nombre **Img_Zipping.conf** pretende ser un algoritmo de compresión de imágenes que, a falta de aportar un novedoso método de compresión general, aprovecha la basta colección de estructuras de datos de la aplicación para guardar información de creación de elementos que, por ello, pueden ser eliminados del archivo comprimido resultante de aplicar el algoritmo que se describe a continuación:

Declaración de variables de almacenado y de compresión de la imagen:

[CLASE_VAR]

I.zipped=0

I.a1r=-1

I.a1g=-1

I.a1b=-1

……

I.a6r=-1

I.a6g=-1

I.a6b=-1

I.a1=0

……

I.a2=0

……

I.a6=0

Funciones de almacenado y marca para compresión:

```
[CLASE_FUNC]
I.a1r=if(I.zipped==0,if({EsHueco(I-1)}==0,{Ficha({Indice({Coordenada(I-
1,X)},{Coordenada(I-1,Y)},1)},C0r)},-1),-1)
I.a1g=if(I.zipped==0,if({EsHueco(I-1)}==0,{Ficha({Indice({Coordenada(I-
1,X)},{Coordenada(I-1,Y)},1)},C0g)},-1),-1)
I.a1b=if(I.zipped==0,if({EsHueco(I-1)}==0,{Ficha({Indice({Coordenada(I-
1,X)},{Coordenada(I-1,Y)},1)},C0b)},-1),-1)
……
I.a6r=if(I.zipped==0,if({EsHueco(I-6)}==0,{Ficha({Indice({Coordenada(I-
6,X)},{Coordenada(I-6,Y)},1)},C0r)},-1),-1)
I.a6g=if(I.zipped==0,if({EsHueco(I-6)}==0,{Ficha({Indice({Coordenada(I-
6,X)},{Coordenada(I-6,Y)},1)},C0g)},-1),-1)
I.a6b=if(I.zipped==0,if({EsHueco(I-6)}==0,{Ficha({Indice({Coordenada(I-
6,X)},{Coordenada(I-6,Y)},1)},C0b)},-1),-1)
I.a1=if(I.zipped==0,if(I.C0r==I.a1r&&I.C0g==I.a1g&&I.C0b==I.a1b,1,0),0
)
……
I.a6=if(I.zipped==0,if(I.C0r==I.a6r&&I.C0g==I.a6g&&I.C0b==I.a6b,1,0),0
)
I.C0r=if(I.zipped==0,if((I.a1+I.a2+I.a3+I.a4+I.a5+I.a6)==0,I.C0r,0),I.C0r)
I.C0g=if(I.zipped==0,if((I.a1+I.a2+I.a3+I.a4+I.a5+I.a6)==0,I.C0g,0),I.C0g)
I.C0b=if(I.zipped==0,if((I.a1+I.a2+I.a3+I.a4+I.a5+I.a6)==0,I.C0b,0),I.C0b)
I.zipped=if(I.C0r==0&&I.C0g==0&&I.C0b==0,1,0)
I.C1r=I.C0r
I.C1g=I.C0g
I.C1b=I.C0b
……
I.C6r=I.C0r
I.C6g=I.C0g
I.C6b=I.C0b
```

Carga de la imagen:

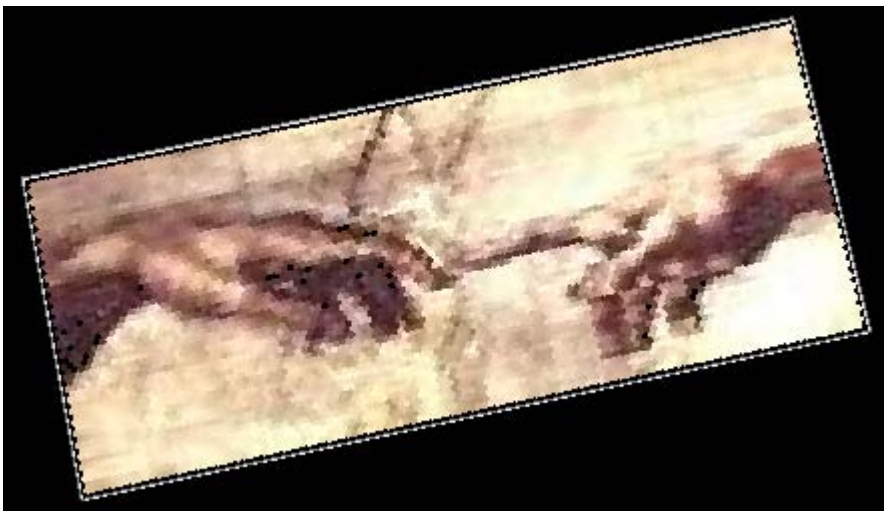
```
[FICHA_DEF]
IMG Sixtina.jpg I 1 1
```

Parada de los elementos ya ejecutados:

```
[CLASE_PARADA]
(I.zipped==1)
```

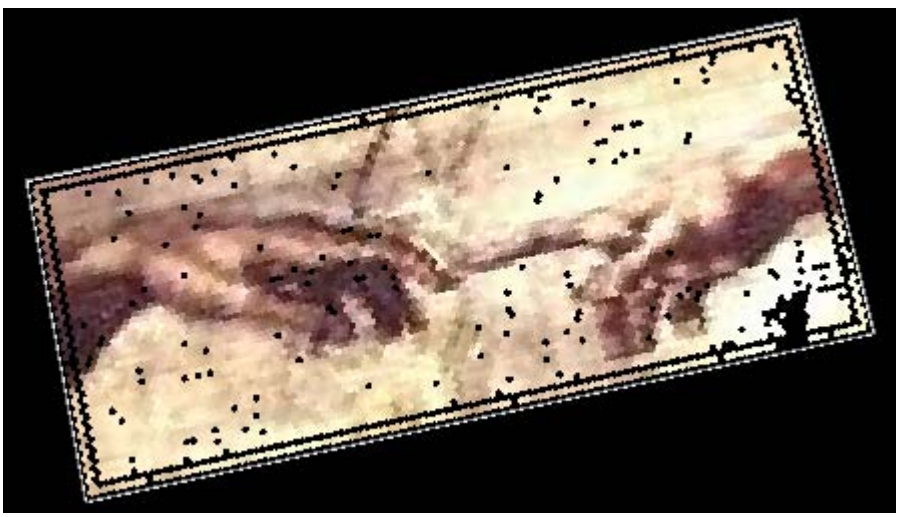
En resumen, el algoritmo examina cada elemento de la imagen buscando algún elemento adyacente al dado con el mismo color. Si existe al menos un elemento entonces se marca el elemento de recorrido de color negro y se almacenan en variables extrínsecas del elemento los colores RGB de cada adyacente del mismo color.

El modelo, por cuestiones de eficiencia en tiempo de ejecución, marca los elementos que encuentran adyacentes que serán eliminados para llegar a comprimir la imagen el máximo posible, tal como se puede observar en la imagen siguiente:



Modelo Img_Zip en paso 0

Una vez calculados los elementos de compresión los marca, tal como se muestra en la siguiente imagen:



Modelo Img_Zipping en paso 1

En el caso de la imagen elegida el ratio de compresión sería bajo, en el caso de las imágenes de galaxias utilizadas se eleva considerablemente. Si se trata de manejar una imagen que se identifique, aunque no sea exacta con el original como pasa con la calidad de envío de archivos por fax o de transmisión de datos que no necesitan un detalle estricto, entonces puede modificarse el modelo para que comprima no solo los elementos con mismo color sino que elimine en la compresión aquellos que tienen un valor de color dentro de un rango, con lo que los niveles de compresión serían bastante elevados.

A partir de aquí, el modelo anterior podría ser preparado para el método inverso que descomprimiera las imágenes y nos devolviera, exacta o aproximadamente, la imagen previa a la compresión, pero para los objetivos de este proyecto no aportaría un nuevo tipo de modelo frente al que aquí se presenta.

Además de este modelo preparado para la descompresión se ha intentado modelar el algoritmo de modificación de imágenes, por su alto, su ancho o por ambos, en tiempo de ejecución, basado en el método de camino de píxeles de mínima energía ideado por el trabajo en colaboración de los autores Shai Avidan y Ariel Shamir a mediados del 2007 y que supone una de las aportaciones mas novedosas al mundo gráfico, especialmente para el ajuste dinámico en contenido Web.

Este modelo excede en parte la capacidad del simulador, en tanto en cuanto no se desarrollen funciones extendidas dedicadas a la gestión de caminos entre elementos adyacentes, grupos conexos de elementos y, probablemente, aquellas específicas para el almacenamiento de resultados previos de operadores basados en funciones recursivas, aunque el proyecto presentado, como se ha indicado en múltiples ocasiones, exhibe parte de su inmensa potencia, por la cualidad inherente de admitir la incorporación de estas nuevas funciones extendidas, posibilitando la simulación de un modelo tan complejo en apariencia.

15 OTROS MODELOS PROPUESTOS

Como final de estos últimos capítulos, dedicados a los distintos tipos de modelos creados, se realizan una serie de propuestas al respecto de otros tipos de simulación o mejoras de las simulaciones presentadas, que pueden ser del interés del lector.

Aunque estos modelos no están reflejados en esta memoria, no podrían incluirse todos los tipos de modelos que el metasimulador es capaz de soportar, son perfectamente factibles mediante el empleo del metasimulador extendido y el conjunto de funciones realizado hasta el momento.

Algunos otros, lo serían en tanto se construyeran e incorporaran en la librería de funciones aquellas que los hicieran posibles.

Una lista de posibles simulaciones sería la siguiente:

- **Combinaciones de modelos**: Se podrían combinar los corales con bandadas de peces o los pájaros volando entre un paisaje de árboles fractales, con tan solo incluir las distintas clases utilizadas en cada uno de los distintos modelos.

- **Fractal Flames**: Este tipo de fractal IFS permite distintas combinaciones de funciones de transformación, a elegir entre las que se desee utilizar. Tratando las funciones de cambio de color según función y valor representado de la transformación pueden obtenerse presentaciones más artísticas de las existentes normalmente.

- **Genetic Algorithms**: Dado que el algoritmo PSO es muy similar al existente para las familias genéticas tras generaciones de ejecución, pueden perfectamente incluirse modelos que generen distintos tipos de seres que vayan mutando con el paso del tiempo.

- **Representación molecular**: Nada impide que el lector intente modelar proteínas, aminoácidos o moléculas conocidas o, inclusive, genere sus propias moléculas. El único límite es la existencia de tan solo 20 elementos adyacentes a uno dado. Los enlaces químicos por debajo de este límite pueden ser representados y mantenerse un control completo sobre las distintas propiedades químicas y físicas de cada molécula o de cada átomo, pudiendo emplear el rango de colores para una mejor visualización de cada molécula representada.

- **Sistemas planetarios**: Al ser pura física, basada en ecuaciones conocidas, no existe problema alguno para crear nuestro propio sistema solar a escala y extenderlo hasta aproximarse al modelo real.

- **Dispositivos mecánicos**: Una vez introducidas las reglas físicas que describen el rozamiento, el choque, la gravedad, los pares de fuerzas y similares, estaríamos en disposición de recrear modelos mecánicos conocidos o simular nuestros propios e innovadores modelos.

- **Juegos**: Al disponer el metasimulador de barras de ajuste de coeficientes y la posibilidad de ejecutarse de forma discreta, podríamos realizar pequeños juegos empleando el movimiento de estos controles para dar al sistema nuestra decisión en cada momento como input a través de los mismos, de forma que el sistema reaccionara respondiéndonos en el siguiente movimiento.

- **Algoritmos recursivos**: El metasimulador representa modelos en un espacio de tres dimensiones. Se pueden plantear modelos de simulación que corran en un plano XYZ para Z = MaxZ, por ejemplo, y que utilicen los planos restantes XYZ' para Z' = 0..MaxZ-1 para desplegar inserciones de elementos que representen aumentos progresivos de la pila de resultados de un típico procedimiento recursivo o los correspondientes borrados de las mismas cuando el proceso esté retornando elementos de la pila hasta volver al punto inicial de la recursión indicando el resultado en el plano XYMaxZ definitivo.

- **Reconocimiento de caracteres**: Una vez dispuesta la funcionalidad adecuada para reconocer un juego implícito de caracteres, podrían ejecutarse modelos que dispusieran fichas en forma de letras manuscritas y que los elementos de reconocimiento, como clases para las distintas letras, comenzaran a aproximar una solución por vecindad sobre los elementos manuscritos, deduciendo un encaje suficiente como para determinar que un carácter a mano pudiera considerarse una letra concreta de dicho juego de caracteres.

- **Serialización de procesos**: Dado un conjunto de elementos que simbolizaran distintos procesos en un sistema operativo con diferentes propiedades que hicieran variar la prioridad de los mismos a la hora de competir por el uso de un recurso, podría simularse el orden exacto de ejecución en el que deberían ser atendidos por ese recurso del sistema.

Y podría seguirse describiendo tipos de modelos casi sin agotar el basto conjunto de posibles representaciones que podrían realizarse, una mayoría con lo existente a la fecha de la entrega de este proyecto, otros mediante la inclusión de nuevas funciones extendidas que posibilitaran su implementación satisfactoria.

16 CONCLUSIONES

Explicar nítidamente la motivación que llevó a la realización de este proyecto siempre ha supuesto una tarea complicada, en tanto en cuanto no existe una respuesta clara para cualquier otra persona distinta del autor. Por otro lado, existe más de un objetivo y, por tanto, hay más de una idea sobre lo que se pretendía lograr. Dado que los objetivos de cualquier proyecto, contrastados con las tareas realizadas, son el constitutivo esencial de las conclusiones objetivas que se pueden ofrecer, se tiene que hablar por objetivos para entender este grupo de conclusiones.

Como ya se comentó al inicio de esta memoria, el primero de los objetivos en cualquier proyecto siempre es subjetivo y pertenece más a la personalidad y gustos del autor que a una necesidad tangible por parte del lector. Un proyecto en el que se mezclen la matemática y la informática es mas un deseo subjetivo que una necesidad real. El segundo de los objetivos particulares, disfrutar con la belleza que reside en la naturaleza e intentar imitarla, lanzó la chispa que puso en marcha todo este proyecto.

Cuando los deseos preliminares se centraron en la idea de hacer de una pantalla un vivero lleno de helechos y se saltó cualitativamente de la idea de escribir sus ecuaciones de crecimiento a poder escribir las ecuaciones de cualquier otro sistema vivo, el objetivo de la simulación generalista emergió con fuerza frente al resto de las metas.

A medida que se satisfacían estos deseos personales, tal como ya se ha reconocido en algunos de los capítulos dedicados a los tipos de modelos de simulación, la propia estructura emergente de la solución inicial planteada no hacía sino sembrar un nuevo interés, un objetivo bien distinto y mucho más concreto, aunque con la misma complejidad a la hora de proponer una explicación al lector.

El ejemplo que puede explicar al lector de la mejor manera posible este nuevo interés es el de la máquina de Turing.

La máquina de Turing es un modelo formal de computador ideado en 1936 por el matemático inglés Alan Turing para responder a la cuestión planteada en 1920 por el matemático alemán David Hilbert al respecto de si las matemáticas son decidibles, modelo con el que demostró que existían problemas que una máquina no podía resolver.

Si nos centramos en la máquina en sí misma, podemos comprobar que esta no es sino un modelo matemático abstracto que formaliza el concepto de algoritmo.

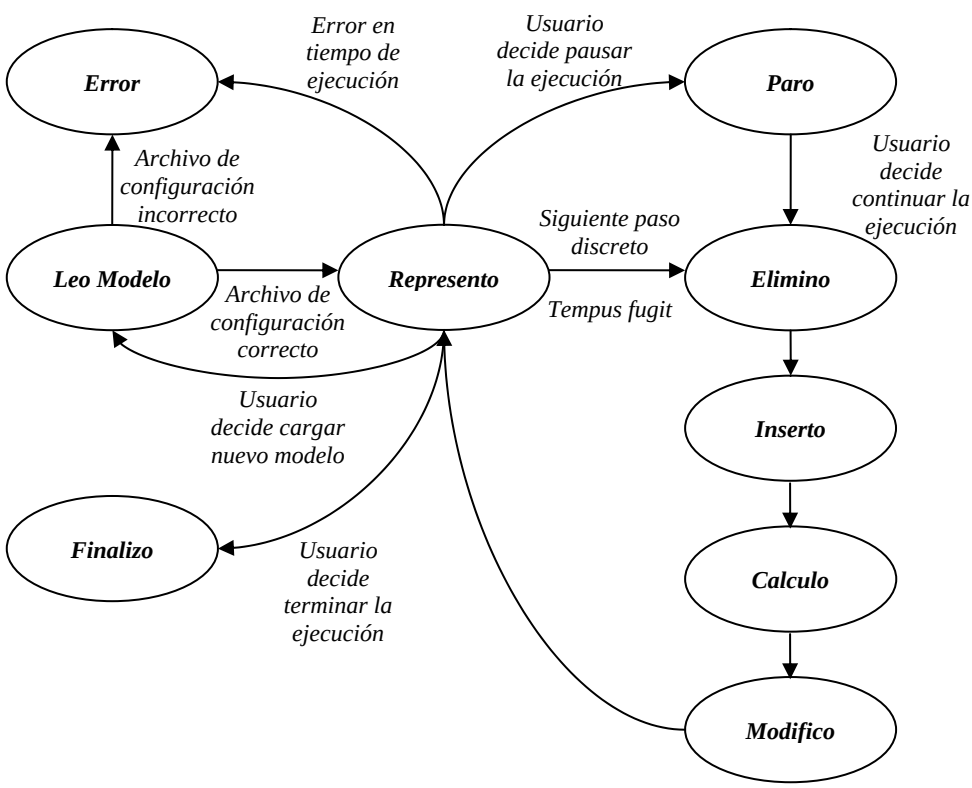
Es en ese punto, resolver cualquier cuestión planteada, donde residen las coincidencias entre el modelo de Turing y este proyecto y la única forma de poder explicar la idea instalada tras los primeros ejemplos basados en modelos naturales.

Como puede leerse en capítulos anteriores, el metasimulador extendido está dotado con un cuerpo de funciones, llamadas funciones extendidas, que amplían la potencia del metasimulador convencional de forma significativa.

El metasimulador convencional nos permite representar figuras de Lissajous, atractores de Lorentz y distintas curvas de ecuación conocida en el espacio de dos o tres dimensiones, efectos de la fuerza de la gravedad sobre partículas con masa o efectos electromagnéticos entre un grupo de partículas con carga y modelos similares basados en ecuaciones lineales sobre dicho espacio. Es decir, cualquier modelo con un sistema de ecuaciones homogéneo es asumible por este sistema.

Si lo pensamos detenidamente, se puede concretar más la definición de diseño de este simulador y decir que representa modelos de ecuaciones con variable en el tiempo o modelos de ecuaciones que utilizan el tiempo de ejecución para discretizar la representación de una curva por sus elementos, dados de uno en uno en cada nuevo valor entero de la variable tiempo de ejecución.

Por tanto, la estructura esencial de ambos simuladores consiste en un inmenso bucle de ejecución y representación con salidas o variantes conocidas, es decir, es un inmenso algoritmo o, si se prefiere, un autómata finito con un número concreto de estados. A saber, los siguientes:



Esto es lógico, ya que se trata de un programa de ordenador, por tanto se trata de un algoritmo o de un autómata con determinados estados.

La cuestión esencial es **¿qué es capaz de simular?**, ya que el modelo convencional sigue el grafo de estados anterior y representa las distintas soluciones de los sistemas de ecuaciones planteados, pero tan solo puede representar ecuaciones matemáticas y no soluciones dadas por algoritmos como, por ejemplo, el juego de vida de Conway.

Si observamos los distintos algoritmos existentes en la literatura comprobaremos que en ellos existen una serie de pasos en los que se determina un nuevo estado de ejecución o se calculan valores hasta lograr un objetivo, momento en el que el algoritmo acaba. El problema fundamental del simulador convencional y los algoritmos conocidos es que las operaciones requeridas por el algoritmo son, la gran mayoría de las veces, bastante más complejas que el cálculo de una expresión matemática convencional. Es aquí donde las funciones extendidas nos permiten satisfacer estos requisitos, mantenidos en una estructura de autómata propia del simulador.

De esta forma, las recetas para ver crecer, desarrollarse o morir elementos en el juego de Conway o representar la evolución de las pinceladas de las abejas virtuales son posibles dentro de este metasimulador extendido. Pero en éste, la cuestión que aparece es otra mucho más compleja: **¿Hasta dónde puede llegar el metasimulador extendido?**

Existe una distinción formal entre algoritmos decidibles e indecidibles, entre autómatas deterministas e indeterministas, entre problemas con complejidad lineal, logarítmica o problemas no resolubles, pero no suele hablarse, como tal, de una jerarquía o tipología de familias de algoritmos, probablemente porque su base común la hace innecesaria. Esta jerarquía ayudaría a identificar las posibilidades funcionales del metasimulador extendido, respondiendo a la pregunta anterior con exactitud. Deberemos entonces, partiendo de la base de la estructura común de un algoritmo, centrarnos en la ejecución de los pasos y en la estructura de estados de cualquier algoritmo para poder trazar la frontera de posibilidades del simulador presentado.

En relación a los pasos de ejecución, lo que podemos calcular o ejecutar, este metasimulador extendido es capaz de lo siguiente:

- Puede calcular el valor de un elemento x, si dicho valor se deduce de una expresión del tipo $x = f(y_0, y_1, \ldots\ldots, y_n)$, donde *f* es una función, en sentido matemático, y el conjunto $\{y_o, y_1, \ldots\ldots, y_n\}$ se compone de variables de elementos conocidos del modelo.

- Puede calcular el valor de un elemento x, si dicho valor se deduce de una expresión del tipo $x = F[p_0, p_1, \ldots\ldots, p_n]$, donde F es una función, en el ámbito de la programación informática, y el conjunto $\{p_0, p_1, \ldots\ldots, p_n\}$ se compone de parámetros de la función que igualan variables de elementos conocidos del modelo y/o valores resultantes de funciones de programación, incluyendo la función F.

- Puede eliminar o añadir nuevos elementos a un grupo e, inclusive, puede hacerlo por subgrupos en vez de elementos simples, siempre que estos se agrupen bajo una condición o propiedad expresable matemáticamente.

- Puede identificar un grupo de elementos común bajo cualquier expresión representable matemáticamente o programáticamente.

- Puede tratar elementos no existentes, es decir, posiciones discretas del entorno de simulación en las que no existe ningún elemento del modelo.

- Puede obviar el cálculo de ciertos elementos, a partir del paso de ejecución en cual se cumpla una condición expresada matemáticamente y/o programáticamente. Por tanto puede concluir la ejecución por falta de elementos, por existir únicamente elementos que no cumplen ninguna condición necesaria para efectuar algún cálculo, por elementos que se encuentran parados o por alguna combinación de las anteriores condiciones.

- Puede responder a una entrada externa, dada por el usuario, y variar su ejecución en función del valor suministrado en cada momento.

Sinceramente, puestas las propiedades una a continuación de otra es difícil saber qué no es capaz de hacer, siempre teniendo en cuenta que sus posibilidades de expansión mediante la ampliación del conjunto de funciones extendidas son casi ilimitadas.

Es justo la búsqueda de esa frontera, la reflexión al respecto de los límites del sistema y, por tanto, la reflexión al respecto de los distintos tipos de algoritmo que podrían ser clasificados, el interés con el que se buscaron modelos que supusieran un reto y pudieran ser evaluados dentro de un tiempo razonable para un proyecto de estas características.

Por ello, parece mejor decantarse por la enumeración de aquellos ejemplos, pensados o buscados, que se han intentado modelizar y que parecen irrealizables con el simulador, para definir los límites del sistema como el total menos este grupo de ejemplos.

Así podemos tomar, como ejemplo representativo para este razonamiento, el modelo basado en el algoritmo de búsqueda de caminos de píxeles de mínima energía, ideado por Shai Avidan y Ariel Shamir para la reducción o ampliación de imágenes sin pérdida de contenido en tiempo de ejecución. Es un ejemplo aparentemente no tratable por el metasimulador extendido.

Este algoritmo realiza una primera etapa en la que identifica aquellos puntos de la recta (x_0,y_{max})-(x_{max},y_{max}) con un valor de energía mínimo y traza sobre los puntos resultantes un camino descendente, uniendo el píxel anterior con el siguiente píxel elegido, formando un hilo de un píxel de ancho, hasta llegar a un punto concreto de la recta (x_0,y_{min})-(x_{max},y_{min}) de forma que al finalizar este paso contaremos con un número de caminos, *vertical seams*, de energía mínima. Tras esto y mediante programación dinámica se elegirá el mejor camino, *optimal seam*, o camino que minimiza el coste dado por la fórmula de energía y será éste el que se eliminará de la imagen, acercando las dos regiones en las que queda dividida esta imagen para hacer desaparecer el vacío existente por la eliminación de este camino. La repetición de estos pasos por cada, pongamos, orden de estrechamiento, hará que la imagen sea corregida dinámicamente pero manteniendo aquellos píxeles que mejor definen el contenido que se muestra en la misma.

Tal como se comentó en el capítulo dedicado a los tipos de modelos basados en gráficos, el intento de simulación del algoritmo *Seam Carver*, que acaba de describirse, fue definido como un tope del sistema, un modelo que la aplicación no podía simular.

Si hacemos una comparación en términos de los pasos existentes en este algoritmo, el metasimulador extendido puede buscar una colección de puntos que sean candidatos a puntos iniciales de los distintos caminos de píxeles de energía mínima, puede recorrer una imagen desde un punto dado y trazar un camino mínimo y entre varios caminos puede decidir cual es el de menor energía, pero no puede llevar a cabo todo esto a la vez, es decir, en el mismo modelo de simulación.

La razón es que el metasimulador, convencional o extendido, no dispone de memoria intermedia de pasos ni siquiera de una serie de meta órdenes que engloben pasos de ejecución sencillos y esta carencia es la que no nos permite realizar este modelo y todos aquellos modelos similares.

En resumen, el límite más evidente parece ser aquel que podríamos definir como **límite de programación** o, en otras palabras, el límite existente en la estructura general del simulador respecto a la ejecución de tareas complejas y distintas en una secuencia dada. Podría decirse que disponemos de una potente rutina, procedimiento, función, como se prefiera llamar, pero no podemos construir un archivo de modelo que requiera la ejecución ordenada de N rutinas de este tipo.

Sin embargo, este límite podría ser rebasado mediante una nueva versión de la aplicación, con un coste no muy excesivo, que contemplara la idea apuntada anteriormente de una secuencia de tareas complejas, constituidas por combinaciones de instrucciones simples como las utilizadas hasta el momento. Un ordinal dado al inicio de cada instrucción permitiría seguir mediante una variable global de número de paso, qué instrucciones simples deben ser ejecutadas en el momento actual y tan solo habría que habilitar condiciones para pasar de una tarea a la siguiente y estructuras temporales que almacenaran los resultados intermedios para que fueran utilizados por el grupo de pasos simples de la siguiente meta orden.

De esta forma, el metasimulador extendido tendría naturaleza de programa, conjunto ordenado de rutinas, y permitiría extender la simulación a todo algoritmo práctico existente, con lo que su potencia limitaría con el tratamiento de problemas no decidibles, es decir, sería tan potente como cualquier solución computacional existente en la actualidad.

En resumen, estaríamos en disposición de crear una segunda versión de este sistema y llamarlo, por ejemplo, **simulador basado en un lenguaje de descripción de modelos**. Pero esto requeriría un proyecto de una magnitud equivalente a los dos metasimuladores existentes juntos, aunque se plantea aquí el interés por disponer de un sistema similar al comentado.

Anexo

La realización del proyecto, desde las reuniones iniciales hasta la finalización del contenido de esta memoria, ha requerido de un tiempo neto considerable que se extiende, en términos de tiempo natural, a lo largo de varios años.

Por este motivo se anexan en esta memoria algunos puntos de interés relativos al metasimulador realizado que, como todo diseño abierto, han ido sucediendo en los últimos meses de proyecto. Aparte de estas anotaciones de última hora, se describirá brevemente el contenido extra adjunto a la memoria para conocimiento del lector.

A.1 MODIFICACIONES FUNDAMENTALES

La búsqueda e implementación de ciertos ejemplos han llevado a la modificación del código fuente del metasimulador extendido que, hasta ese momento, se presuponía como solución final.

Si bien existen bastantes diferencias entre la base común original, compartida por el metasimulador convencional y este simulador, algunas de ellas son sustanciales frente al diseño de partida y merecen una consideración aparte, ya que dotan a este metasimulador de una identidad funcional exclusiva.

A.1.1 Ejecución ordenada de funciones:

Al inicio de las pruebas de la base común implementada en este metasimulador, los ejemplos de naturaleza matemática, basados en ecuaciones homogeneas, arrojaban ejecuciones en las que pasaba inadvertido el problema que impediría, mas adelante, el resto de las simulaciones.

Cuando se solicita al metasimulador que calcule el siguiente punto (X,Y,Z) al que se desplazará una ficha que representara cualquier partícula convencional, el sistema terminará por darnos cada uno de los 3 valores que componen la nueva posición tridimensional.

Pero el hecho de que el modelo no lo requiera nos evita plantearnos una cuestión trascendental: **¿En que orden nos entrega estos valores?** O, dicho de otra forma, **¿Qué función se calcula primero?** La respuesta exacta sería, la primera función según el orden de la tabla hash donde se almacenan las funciones.

Si ahora intentáramos simular el comportamiento de los elementos del juego de vida, en vez de mover partículas según un vector de desplazamiento, comprobaremos que el sistema es incapaz de representarnos el algoritmo de Conway.

Precisamente por eso, por ser un algoritmo, es decir, una secuencia ordenada de instrucciones.

Tal como sucedió con los primeros fracasos de la simulación del juego de vida, el hecho de que el metasimulador coja la siguiente función que encuentre en la tabla hash de funciones y no la función que el autor del modelo necesite en cada momento, es algo inaceptable para las simulaciones que este programa debería realizar.

Tras comprobar este efecto en las primeras simulaciones extendidas, se tuvo que implementar un sistema ordenado de ejecución sobre la base existente que posibilitara esta característica sin afectar al resto de la funcionalidad. La solución utilizada se basa en un vector ***OrderF*** que almacena cada función del modelo en el orden de escritura, tal como haría un compilador con las líneas de código de cualquier programa, de forma que en la función principal de cálculo, ***doF()***, todo se supedite a un bucle de recorrido de este vector para que sea esta estructura la que nos indique las funciones a tratar, en el orden exacto en que se requieren por parte de la simulación dada.

En esquema, la modificación anterior sería como sigue:

```
public class Espacio {
        // Atributos de la clase
        …
        String[] OrderF
        …

        //Métodos de la clase
        …
        private void doF(Hex h) {
                …
                int i = 0;

                while(OrderF[i] != null) {
                        String idF;
                        …
                        idF = OrderF[i];
                        …
                }
        }
        …
}
```

Gracias a esta modificación, el autor del modelo escribe cada expresión funcional en el orden en el cual desea que sea ejecutada por el metasimulador extendido.

A.1.2 Filtro para gráficos:

Tal como ya se comentó en el capítulo dedicado a modelos gráficos, la simulación basada en imágenes, según la base común de partida, obligaba a almacenar centenares de miles de datos de cada una de las miles de fichas que componen la imagen dada.

Esto hace que, salvo para presentar algunas imágenes y realizar alguna simulación con ciertas imágenes muy pequeñas, el sistema no sea capaz de gestionar ese volumen de información y ejecutar el modelo solicitado.

Para evitar este problema en la medida de las posibilidades existentes hasta el momento, se decidió realizar una modificación posterior que consiste en filtrar la creación de fichas con un rango de color próximo al negro absoluto en una medida determinada.

Tal como ya quedó explicado en el apartado referido anteriormente, un valor de corte de **RGB = [40,40,40]**, es decir, despreciar la orden de creación de ficha para aquellas con un color RGB que se encuentre entre el negro puro, RGB = [0,0,0], y el valor de corte produjeron una reducción drástica del volumen de información a mantener.

Las imágenes de galaxias, empleadas mayoritariamente para las pruebas, vieron reducido su volumen de datos a tan solo un 8% del total de la información, con una calidad de imagen indistinguible de la original.

Además, la reducción de datos es consecuencia directa de una reducción del número de fichas, por tanto del número de elementos cuyas funciones deben ser gestionadas, lo que significa, evidentemente, una mayor velocidad a la hora de simular el modelo gráfico.

A.1.3 Reducción de memoria:

Una de las modificaciones más importantes, en términos de eficiencia de la solución, es la realizada en las estructuras de información que relacionan el mundo matemático con el gráfico.

Como se ha podido leer en los capítulos dedicados al diseño de la solución, se propusieron dos estructuras, **Id2XYZ** y **XYZ2Id**, cuyo cometido es almacenar las relaciones de índice de elemento y posición, de la siguiente forma:

*Sea **h** un elemento de la simulación con índice **i** y coordenadas (**a,b,c**), entonces:*

- *Id2XYZ(**i**) = "**abc**"*
- *XYZ2Id[**a,b,c**] = **i***

De esta forma el metasimulador puede conocer el elemento que se encuentra en una posición determinada del espacio, o si no existe elemento en esa posición, y obtener la posición de un elemento por su índice.

Estas estructuras se encuentran por todo el desarrollo de la base común y, por tanto, de ambos simuladores.

Así mismo, se creó una estructura rejilla en la que la parte gráfica anotaba los elementos a pintar, eliminar, mover o colorear en cada paso.

Al final de cada paso de ejecución, una vez se han calculado los nuevos valores de cada uno de los elementos en la parte matemática, esta estructura se recorría por el total de sus 3 dimensiones para anotar los distintos cambios y que estos se representaran gráficamente antes de comenzar el siguiente paso de ejecución.

La forma de almacenamiento y gestión de esta estructura es la siguiente:

> *Sea* $\boldsymbol{H}$*, el conjunto* $\{h_1, h_2, .., h_n\}$ *con los* $\boldsymbol{N}$ *elementos de la simulación que existen en un momento de ejecución dado, entonces:*
>
> - *Si* $h_i.status = inserted$*, entonces* $rejilla[h_i.X][h_i.Y][h_i.Z] = h_i$
> - *Si* $h_i.status = deleted$*, entonces* $rejilla[h_i.X][h_i.Y][h_i.Z] = null$
> - *Si* $h_i.status = moved$*, entonces:*
>
> $$rejilla[h_i.prevX][h_i.prevY][h_i.prevZ] = null$$
> $$rejilla[h_i.X][h_i.Y][h_i.Z] = h_i$$
>
> *Leer* $h_{ijk} = rejilla[i][j][k]$*, para todo i,j,k y hacer:*
>
> - $h_{ijk}.status = inserted$ *-> pintar nuevo prisma en (i,j,k)*
> - $h_{ijk}.status = deleted$ *-> borrar prisma en (i,j,k)*
> - $h_{ijk}.status = moved$ *-> borrar prisma en (i',j',k') y pintarlo en (i,j,k)*
> - $h_{ijk}.status = colored$ *-> colorear prisma en (i,j,k)*

A medida que se creaban los modelos gráficos o al ampliar el espacio de representación en algún modelo anterior, para que los pájaros volaran a través de un "cielo" más grande por poner un ejemplo, el consumo de memoria de estas estructuras crecía de tal forma que hacía inmanejable determinados modelos.

Para hacerse mejor idea del consumo referido, veamos una tabla de valores:

Modelo	Espacio	Casillas	Fichas	Id2XYZ	XYZ2Id	Rejilla	Huecos	Eficiencia
Coral	100*100*100	1.000.000	50	50	1.000.000	1.000.000	999.950	**0.005 %**
Boids	300*300*300	9.000.000	15	15	9.000.000	9.000.000	8.999.985	**0.00016 %**
Boids	500*500*500	125.000.000	15	15	125.000.000	125.000.000	124.999.985	**0.000012 %**

Estos datos tan solo calculan la eficiencia entre el número de elementos de una simulación y una de las estructuras, porque la relación de gasto de elementos frente al total de estructuras arrojaría unos porcentajes bastante más bajos.

De cualquier forma, estas cifras dejan en evidencia a la solución cuando esta debe enfrentarse a la simulación de modelos con espacios de representación basados en cubos con cientos de casillas de lado.

Para evitar este excesivo consumo de memoria que, según modelos, puede provocar el fallo en la simulación, o siquiera su carga, se crearon dos nuevas estructuras:

idxyz, de tipo Vector, tal que:

- *Sea* $\mathbf{H = \{h_1, h_2, .., h_n\}}$*, entonces idxyz.size() =* N
- *Sea* $\mathbf{h_i}$ *con índice i en posición (**a,bc**), entonces*
  *idxyz[**i**] = "**abc**"*
  *idxyz.getId(**a,b,c**) = **i***
- *Sea la posición (**a,bc**) **vacía**, entonces idxyz.existsxyz(**a,b,c**) = **false***

rejilla, de tipo **Cubo**, clase compuesta de:

malla, de tipo Vector
posiciones, de tipo Vector
posX, posY, posZ, de tipo int

Sea $\mathbf{H}$*, el conjunto* $\mathbf{\{h_1, h_2, .., h_n\}}$ *con los* $\mathbf{N}$ *elementos de la simulación que existen en un momento de ejecución dado, entonces:*

- *Si* h_i*.status = inserted, entonces* h_i *= Cubo.getFicha(*h_i*.X,*h_i*.Y,*h_i*.Z)*
- *Si* h_i*.status = deleted, entonces Cubo.positionHaveFicha(*h_i*.X,*h_i*.Y,*h_i*.Z) = false*
- *Si* h_i*.status = moved, entonces:*
  Cubo.resetPosition(h_i*.prevX,*h_i*.prevY,*h_i*.prevZ)*
  h_i *= Cubo.getFicha(*h_i*.X,*h_i*.Y,*h_i*.Z)*

Leer h_i *= Cubo.getFicha(*h_i*.X,*h_i*.Y,*h_i*.Z), para todo* h_i *perteneciente a* $\mathbf{H}$ *y hacer:*

- h_i*.status = inserted -> pintar nuevo prisma en (*h_i*.X,*h_i*.Y,*h_i*.Z)*
- h_i*.status = deleted -> borrar prisma en (*h_i*.X,*h_i*.Y,*h_i*.Z)*
- h_i*.status = moved -> borrar prisma en (*h_i*.prevX,*h_i*.prevY,*h_i*.prevZ)*
  y pintarlo en (h_i*.X,*h_i*.Y,*h_i*.Z)*
- h_i*.status = colored -> colorear prisma en (*h_i*.X,*h_i*.Y,*h_i*.Z)*

De esta forma, solo se consume memoria para almacenar y gestionar los elementos de la simulación en un momento dado de la ejecución, lo cual redunda en la eficiencia de la solución de una forma muy significativa.

Si se observan los nuevos datos mos trados en la tabla siguiente:

Modelo	Espacio	Casillas	Fichas	idxyz	Rejilla	Huecos	Eficiencia
Coral	100*100*100	1.000.000	50	50	50	0	**100 %**
Boids	300*300*300	9.000.000	15	15	15	0	**100 %**
Boids	500*500*500	125.000.000	15	15	15	0	**100 %**

se puede comprobar que la mejora es, aparte de evidente, absoluta. El metasimulador extendido pasa de estar imposibilitado para simular 15 boids volando en un cubo de 500 posiciones de lado a poderlo hacer sin mayor problema y con un ahorro de memoria sustancial, al consumir tan solo lo necesario para cada elemento del modelo y no requerir un almacenado de 125 millones de posiciones, de las que todas, salvo 15 de ellas, están vacías y sin uso alguno.

A.2 CD DE RECURSOS

Dada la naturaleza del proyecto, basado en un programa de simulación y una serie de modelos de ejemplo, se ha incluido como anexo a esta memoria un CD con recursos relativos a la solución, que complementan la memoria y mediante los cuales el lector puede visualizar los ejemplos en acción o instalar en su máquina la aplicación para generar sus propios modelos de prueba.

El CD entregado junto con la memoria contiene las siguientes carpetas:

\Instalacion
: Archivos necesarios para instalar la aplicación en el ordenador del usuario. Contiene los ejecutables de la solución en modo línea de comandos o en modo win32 de MS-Windows en cualquiera de las versiones actuales.

\Codigo
: Archivos .java con el código fuente del proyecto.

\Modelos
: Archivos .conf con los distintos modelos realizados en el proyecto.

\Memoria
: Archivo .pdf con esta memoria

\Videos
: Archivos .wmv con vídeos basados en capturas de pantalla de las distintas ejecuciones de los modelos, de forma independiente, un vídeo con la ejecución continuada de todos los modelos y un último vídeo, similar al anterior, pero en una versión resumida y comentada de menor duración.

En cualquiera de las carpetas, el lector podrá encontrar un archivo **leeme.txt** en el que se darán las oportunas indicaciones de uso, los distintos pasos de instalación, se realizarán los comentarios de versión actualizados o se incluirán breves descripciones sobre el contenido existente en cada momento.

Bibliografía

La naturaleza del proyecto realizado hace que este apartado de referencias bibliográficas sea mas un compendio de recursos de Internet que una lista de libros técnicos, pero donde puede hacerse referencia al título de un libro y su autor puede incorporarse una dirección de Internet y el autor o grupo que compila el conocimiento empleado para la realización de este simulador.

Las referencias utilizadas en el diseño y desarrollo de este proyecto, divididas por áreas del conocimiento, se detallan en los apartados descritos a continuación.

B.1 Informática

Compiladores. Principios, técnicas y herramientas
A.V. Aho, R. Sethi y J.D. Ullman
Addison Wesley Longman, 1990

Programación en Java
A. E. Walsh
Anaya multimedia, 1997

Borland Jbuilder 2006 Enterprise
http://www.codegear.com/products/jbuilder
Borland Software Corporation, 2006

Java 3D
http://java.sun.com/products/java-media/3D/
Sun Microsystems, 2007

JEP, Java Math Expression Parser
http://www.singularsys.com/jep/
Singular Systems, 2007

B.2 Matemáticas

Dios creó los números.
Los descubrimientos matemáticos que cambiaron la historia
S. Hawking
Crítica, 2006

Curvas y Superficies
http://www.epsilones.com/paginas/i-curvas.html
Alberto Rodríguez Santos
Epsilones, 2007

3D Graphic Equation Calculator
http://www.algebrahelp.com/calculators/equation/3dgraphing/pg2.htm
jrmski
Algebra.help, 2000

Basic 3D Math
http://www.geocities.com/SiliconValley/2151/math3d.html
Mark Feldman
The Win95 Game Programmer's Encyclopedia, 1997

Sing Surf
A Program for calculating singular algebraic curves and surfaces
http://www.singsurf.org/singsurf/SingSurf.html
Richard Morris, 2005

B.3 Tipo I. Naturaleza: Corales, Vegetales y Esponjas

La Proporción Áurea.
La historia de PHI, el número más sorprendente del mundo
Mario Livio
Ariel, 2006

Algorithmic Botany. Papers
http://algorithmicbotany.org/papers/#abop
Biological Modeling&Visualization research group. Dpt. of Computer Science
University of Calgary

Modelado y Visualización de formaciones de coral
R. Quirós, X. Lluch, M.J. Vicent, J. Huerta.
Dto. de Informática.
Universidad Jaime I de Castellón.

The algorithmic beauty of plants
P. Prusinkiewicz, A. Lindenmayer
New York, Ed. Springer-Verlag, 1990

B.4 Tipo II. Hex Life Game: La Vida en un Prisma Hexagonal

Ruedas, Vida y otras diversiones matemáticas
Martin Gardner
Labor, 1985

3D Game of Life
http://www.ibiblio.org/e-notes/Life/Game.htm
Carter Bays
E-Notes, 2000

Conway's Game of Life
http://en.wikipedia.org/wiki/Conway's_Game_of_Life
Wikipedia, 2007

B.5 Tipo III. Vants: Las Abejas Virtuales

The Computational Beauty of Nature
Computer Explorations of Fractals, Chaos, Complex Systems & Adaptation
Gary William Flake
Bradford Books, The MIT Press, 2000

B.6 Tipo IV. Particle Swarm Optimization: Sincronía entre pájaros

Boids
http://www.vergenet.net/~conrad/boids/
Conrad Parker
Vergenet, 2007

Boids
Flocks, Herds and Schools. A distributed behavioral model
http://www.red3d.com/cwr/boids/
Craig Reynolds
Red3d, 2001

Using Particle Swarm Optimization for offline training in a racing game
http://www.gamasutra.com/features/20051213/villiers_01.shtml
CMP Media LLC
Gamasutra, 2005

B.7 Tipo V. IFS y Chaos Game: Sierpinski, Barnsley y el Árbol de la manzana

Fractales en la red
http://matap.dmae.upm.es/cursofractales/index.htm
Bartolo Luque, Aida Agea
Dto. Matemática aplicada y Estadística
Universidad Politécnica de Madrid, 2007

L-system
http://en.wikipedia.org/wiki/L-systems
Wikipedia, 2007

The Fractal Flame algorithm
http://flam3.com/flame.pdf
Flam3.com, 2007

B.8 Tipo VI. Gráficos: Píxeles hexagonales

Seam Carving for content-aware image resizing
http://www.faculty.idc.ac.il/arik/imret.pdf
Shai Avidan, Ariel Shamir
The Interdisciplinary Center Efi Arazi of Computer Science, 2007