24 | Proyecto: El juego de la vida

Autor original: José Galaviz Casas
Manuel Alcántara Juárez
Verónica Esther Arriola Ríos

Prerrequisitos

- Arreglos
- Listas y genéricos
- Herencia

Meta

Que el alumno desarrolle la lógica de una aplicación para un tema real haciendo uso de orientación a objetos y arreglos.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Manejar con fluidez los conceptos de **clase** y **objeto**.
- Utilizar herencia para el diseño de una aplicación que reutiliza código haciendo uso de una jerarquía de clases.
- Crear una animación en `JavaFX` gobernada por una simulación matemática.

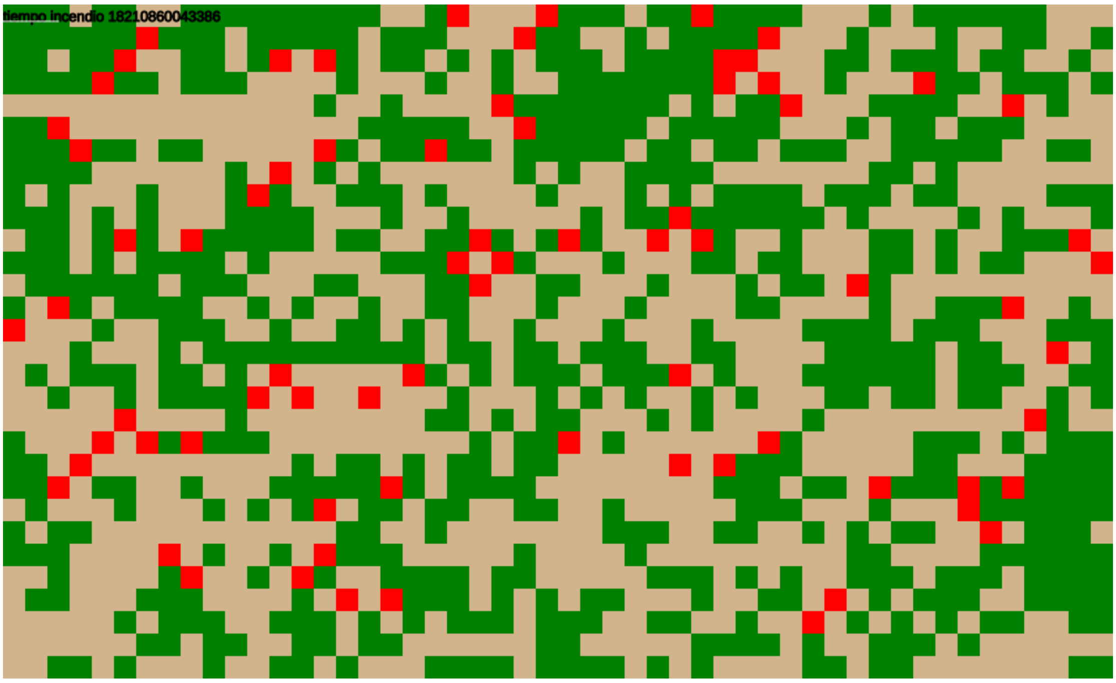

Figura 24.1 Autómata celular que simula propagación de incendios. El sistema está representado por una malla de células cuadradas. Las células café representan tierra, las verdes, árboles y las rojas árboles incendiados.

Antecedentes

En términos generales un sistema dinámico es un modelo que describe el comportamiento, a lo largo del tiempo, de un sistema en función de sus estados previos. Un caso particular de un sistema dinámico son los sistemas dinámicos discretos, en los que el tiempo no es una variable continua sino que avanza a "pasos". En general un sistema dinámico discreto está descrito completamente por la función $f : \mathbb{X} \to \mathbb{X}$ que permite determinar el estado actual del sistema x_t en función de su estado previo x_{t-1}, es decir $x_t = f(x_{t-1})$, donde x_{t-1} y x_t son dos estados de un conjunto de posibles estados del sistema $\mathbb{X}$, en notación: $x_{t-1}, x_t \in \mathbb{X}$.

Un autómata celular es un sistema dinámico discreto, en el que de hecho tanto el espacio como el tiempo y los posibles estados del sistema son discretos. El espacio se divide en *células* regulares de igual tamaño y forma, lo que constituye una *malla* Figura 24.1. En cualquier instante del tiempo t cada célula de la malla posee un valor determinado de un conjunto de posibles valores, lo que constituye su *estado*. Se comienza definiendo el estado inicial x_0 en forma determinista o aleatoria. Para determinar el estado de la célula en el siguiente paso temporal (en $t+1$) se aplica una *regla local*, una función que determina, con base en el valor de la célula misma y de algunas de sus células vecinas al tiempo t, el nuevo valor del estado de la célula.

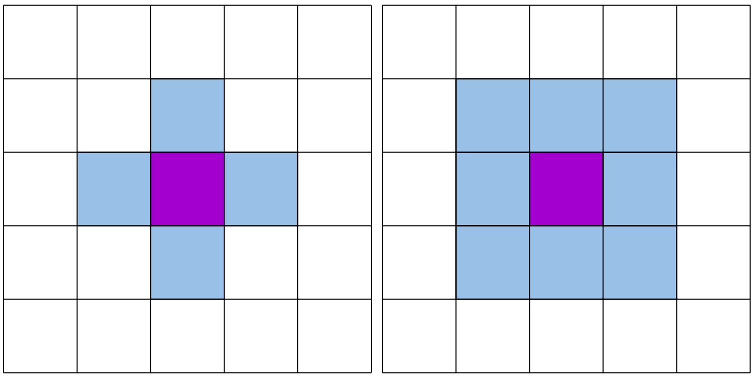

Figura 24.2 Izquierda: vecindad de Von Neumann. Derecha: vecindad de Moore. Los dos tipos de vecindades más usuales en autómatas celulares bidimensionales. El estado de la célula central es determinado, en el siguiente paso temporal, por su valor actual y el de las células coloreadas a su alrededor.

Por supuesto, dado que el estado de una célula está en función de su estado previo y del estado previo de sus células vecinas, conviene especificar quiénes son sus células vecinas. Hay una infinidad de opciones, pero las más usuales en los autómatas celulares bidimensionales son dos tipos de vecindades:

Von Neumann La vecindad de Von Neumann de una célula en la posición (i, j) está constituida por aquellas células que se encuentran arriba $(i, j-1)$, abajo $(i, j+1)$, a la derecha $(i+1, j)$ y a la izquierda $(i-1, j)$ de la célula en cuestión.

Moore La vecindad de Moore, definida por John Conway, considera además las células que están arriba a la derecha, arriba a la izquierda, abajo a la derecha y abajo a la izquierda.

En la Figura 24.2 se ilustran los dos tipos de vecindad. Una forma de definir la malla es de tal modo que **todas** las células tengan igual número de vecinos. Para conseguirlo, las células superiores tienen como vecinos en la parte de arriba a las células que se encuentran en la parte inferior y lo mismo para las células que se encuentran en los extremos izquierdo y derecho. Podría decirse que la malla está doblada en forma de dona.

Los autómatas celulares han recibido mucha atención, porque han resultado ser modelos muy adecuados para describir el comportamiento de algunos fenómenos naturales muy complejos, con unas cuantas reglas locales muy simples. Desde el punto de vista teórico resultan ser también muy interesantes, dado que son un ejemplo de sistemas con interacciones no lineales: el conocer cómo operan locamente no permite muchas veces conocer el comportamiento general del autómata a largo plazo, son reglas simples que engendran comportamientos complejos, algo que suele calificarse como *propiedades emergentes*. Los autómatas celulares resultan estar muchas veces emparentados con los procesos caóticos o en la frontera del caos, con fenómenos de criticalidad auto-organizada y en general con lo que se denominan sistemas complejos.

El juego de la vida

El *juego de la vida* es el mejor ejemplo de un autómata celular, diseñado por el matemático británico John Horton Conway en 1970.

Desde un punto de vista teórico, es interesante porque es equivalente a una máquina universal de Turing, es decir, todo lo que se puede computar algorítmicamente se puede computar en el juego de la vida.

Desde su publicación, ha atraído mucho interés debido a la gran variabilidad de la evolución de los patrones. Se considera que la vida es un buen ejemplo de emergencia y auto organización. Es interesante para los científicos, matemáticos, economistas y otros observar cómo patrones complejos pueden provenir de la implementación de reglas muy sencillas. Para muchos aficionados, el juego de la vida sólo era un desafío de programación y una manera divertida de usar ciclos de la CPU. Para otros, sin embargo, el juego adquirió más connotaciones filosóficas. Desarrolló un seguimiento casi fanático a lo largo de los años 1970 hasta mediados de los 80.

El juego de la vida es en realidad un juego de cero jugadores, lo que quiere decir que su evolución está determinada por el estado inicial y no necesita ninguna entrada de datos posterior. El *tablero de juego* es una malla formada por cuadrados (*células*) que se extiende por el infinito en todas las direcciones. Cada célula tiene 8 células vecinas, que son las que están próximas a ella, incluso en las diagonales (vecindad de Moore). Las células tienen dos estados: están *vivas* o *muertas* (o *encendidas* y *apagadas*). El estado de la malla evoluciona a lo largo de unidades de tiempo discretas (se podría decir que por turnos). El estado de todas las células se tiene en cuenta para calcular el estado de las mismas al turno siguiente. **Todas las células se actualizan simultáneamente**.

Las transiciones dependen del número de células vecinas vivas:

- Una célula muerta con exactamente 3 células vecinas vivas *nace* (al turno siguiente estará viva).
- Una célula viva con 2 ó 3 células vecinas vivas sigue viva, en otro caso *muere* o permanece muerta (por *soledad* o *superpoblación*).

Existen numerosos tipos de patrones que pueden tener lugar en el juego de la vida, como patrones estáticos “vidas estáticas”, patrones recurrentes “osciladores” y patrones que se trasladan por el tablero “naves espaciales”. Los ejemplos más simples de estas tres clases de patrones se muestran en la Figura 24.3. Las células vivas se muestran en negro y las muertas en blanco. Los nombres son más conocidos en inglés, por lo que también se muestra el nombre de estas estructuras en dicho idioma.

El bloque y el barco son vidas estáticas, el parpadeador y el sapo son osciladores y el planeador y la nave espacial ligera son naves espaciales que recorren el tablero a lo largo

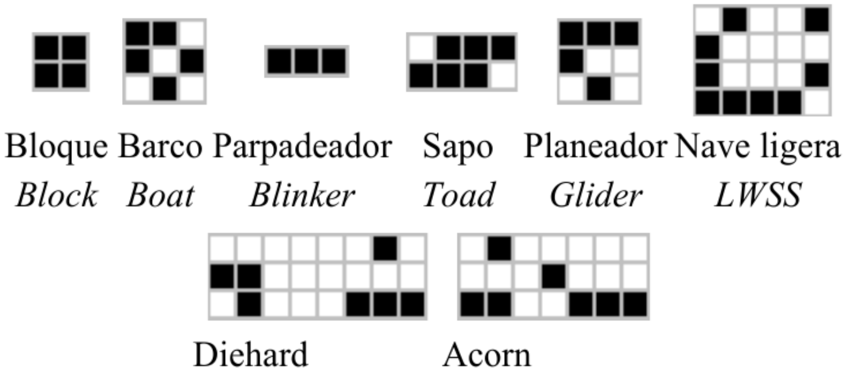

Figura 24.3 Patrones en el juego de la vida.

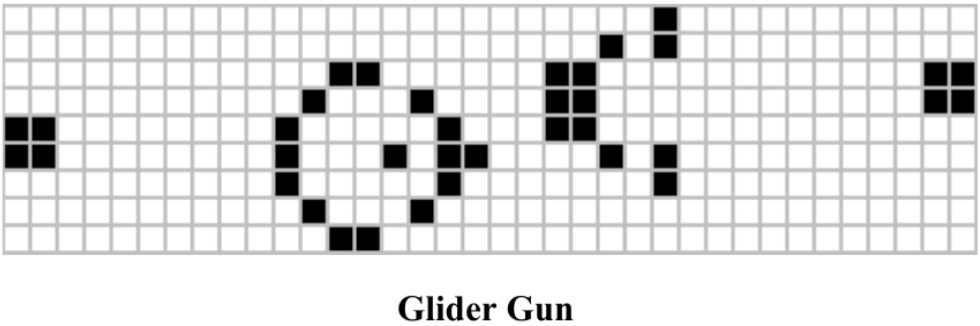

Figura 24.4 Patrón tipo planeador en el juego de la vida.

del tiempo. El primer planeador que se ha descubierto sigue siendo el más pequeño que se conoce: *Glider Gun* Figura 24.4

Los patrones llamados "Matusalenes" pueden evolucionar a lo largo de muchos turnos, o generaciones, antes de estabilizarse. El patrón "Diehard" desaparece después de 130 turnos, mientras que "Acorn" tarda 5206 turnos en estabilizarse en forma de muchos osciladores y en ese tiempo genera 13 planeadores.

En la aparición original del juego en la revista, Conway ofreció un premio de 50 dólares por el descubrimiento de patrones que crecieran indefinidamente. El primero fue descubierto por Bill Gosper en noviembre de 1970. Entre los patrones que crecen indefinidamente se encuentran las "pistolas", que son estructuras fijas en el espacio que generan planeadores u otras naves espaciales; "locomotoras", que se mueven y dejan un rastro de basura y "rastrillos", que se mueven y emiten naves espaciales.

JavaFX

Arquitectura del proyecto

En este proyecto tendrás que programar tanto la lógica de algunos autómatas celulares, como su visualización gráfica. El código auxiliar que acompaña este proyecto ya tiene un demo ejecutable, que podrás extender para completar esta tarea. En el archivo *README.md* encontrarás instrucciones más detalladas sobre cómo ejecutarlo.

Para programar una aplicación de `JavaFX` se debe comenzar con tres elementos:

`Application` La clase principal debe extender `javafx.application.Application`. La clase base se encargará de toda la configuración previa al lanzamiento de la interfaz de usuario, incluyendo todas las llamadas al sistema operativo, que se esconden detrás. El código correspondiente a tu programa comenzará a ejecutarse a partir del método `public void start(Stage primaryStage) throws Exception`, que debes sobreescribir.

En el código auxiliar, la clase `automatas.Demo` cumple esta función.

`Stage` El método `start` de la clase `Application` recibe como parámetro un *escenario*. Esta clase representa al lugar donde se montará tu *escena*. Una aplicación puede tener varias escenas y éstas se van representando en el escenario, puedes alternar entre escenas, en forma análoga a lo que sucede en un teatro o una película.

`Scene` La *escena* es el contenedor para todos los objetos que se mostrarán al usuario, con sus animaciones.

Para este proyecto sólo se usará una escena, pero se modificará su contenido dependiendo del autómata que se quiera ejecutar. Se crea en la clase `Demo` y mide 800×600 pixeles.

Nodos

En `JavaFX` todos los elementos que aparecen en la pantalla se agregan a una escena y son objetos de tipo `Node`, incluso la cámara que *filma* la escena, aunque no la veamos. La Figura 24.5 muestra los elementos más importantes de la jerarquía de clases.

Hay tres tipos de nodos que te interesarán particularmente:

`Parent` Estos nodos sirven para contener a otros nodos. Varias subclases de `Parent` ayudan a colocar a los nodos en posiciones específicas dentro de la ventana. Por ejemplo, un `BorderPane` permite colocar componentes fácilmente en las regiones superior, centro, inferior, izquierda y derecha. Un `VBox` permite colocar un elemento debajo de otro. Un `Group` permitirá colocar ciertos componentes en coordenadas específicas.

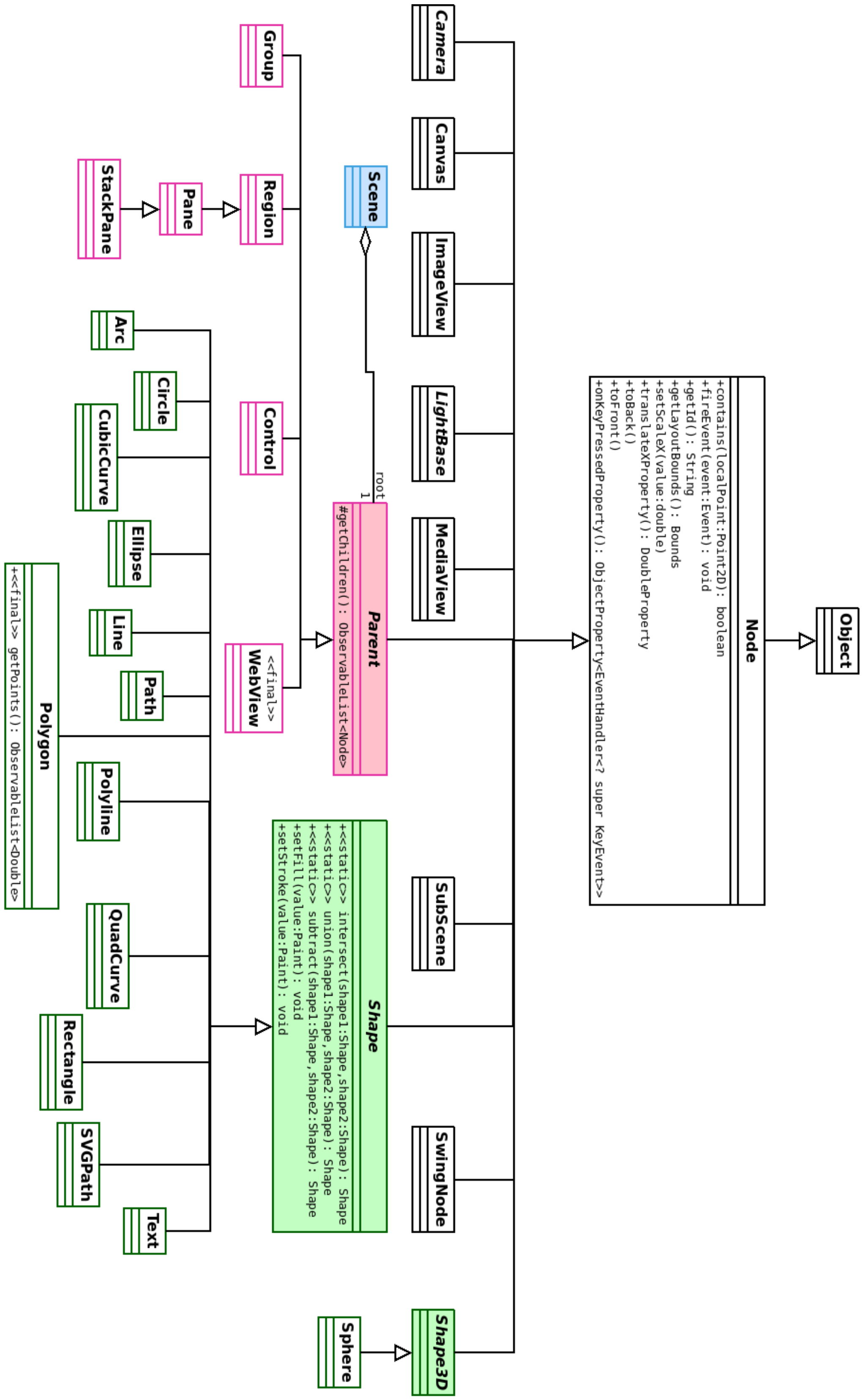

Figura 24.5 Diagrama de clases UML para los nodos de JavaFX.

`Shape` Dibujan figuras geométricas. El código auxiliar usa como ejemplos `Line` y `Rectangle`.

`Control` Los controles ejecutan código cuando el usuario interactúa con ellos. Ejemplos son `Button` y `MenuItem`, también se usan en el código auxiliar.

AnimationTimer

Un `AnimationTimer` es un objeto particular que implementa un temporizador. Su método `handle` es invocado automáticamente en forma periódica para dibujar el contenido de la escena cada cuadro.

La clase `Autómata` del ejemplo extiende `AnimationTimer` de modo que el código en su método `handle` se ejecutará para simular cada paso en la evolución de un autómata.

Ejercicios

Hay varios modelos basados en autómatas celulares que podemos programar, todos tienen varios atributos y procedimientos en común, por lo que es conveniente usar herencia para implementarlos. Se propone que elijas algunos de los modelos siguientes y los implementes extendiendo la clase `Autómata`.

Opción 1: Actividad sísmica

Se utilizan vecindades de Von Neumann. Cada célula puede tomar valores en el conjunto $\mathbb{X} = \{0, \ldots, M\}$. A la constante M se le llama *valor umbral*.

1. En $t = 0$ la malla se inicializa con valores aleatorios de entre los valores válidos inferiores a M.

2. En cada paso temporal t se elige aleatoriamente un sitio de la malla y su valor se incrementa en una unidad.

   Otra opción es incrementar los valores de todas las células, esta última opción evoluciona más rápido.

3. Si alguna célula alcanzó el valor umbral en $t - 1$, para t decrementa su valor en cuatro unidades.

4. Si un sitio tiene vecinos en valor umbral en $t - 1$, para t incrementa su propio valor en una unidad por cada vecino en valor umbral (sin rebasar el valor umbral).

Los sitios (células) de la malla con valor umbral modelan aquellos lugares de una falla tectónica que han acumulado mucha energía y ya no pueden más. Cuando un sitio de la falla alcanza el umbral lo único que le queda por hacer es romperse liberando energía que deben absorber, si pueden, sus sitios vecinos. Cuando muchos sitios se rompen al mismo tiempo se genera un sismo, la magnitud de éste está en función del número de células que se rompen.

Opción 2: Un modelo de propagación de epidemias

Se utilizan también vecindades de Von Neumann. Los valores de las células están en el conjunto $\mathbb{X} = \{0, \ldots, a + g\}$, dividido en tres subconjuntos: $A = \{0\}$, $B = \{1, \ldots, a\}$ y $C = \{a + 1, \ldots, a + g\}$, el conjunto A es *susceptible*, el B es *infeccioso* y el C es *inmune*.

1. Un individuo susceptible se vuelve infeccioso si al menos un vecino es infeccioso.
2. Un individuo inmune después de g pasos se vuelve susceptible.
3. Un individuo que ha sido infeccioso o inmune por menos de g pasos sólo incrementa su valor.
4. Un individuo infeccioso después de a pasos se vuelve inmune.
5. Un individuo susceptible sin vecinos infecciosos permanece susceptible.

Opción 3: Un modelo de incendios forestales

El modelo que se describe a continuación se suele llamar incendio forestal estocástico.

Los valores de las células son 0 (*vacío*), 1 (*árbol*) y 2 (*árbol incendiado*), se utilizan vecindades de Von Neumann. Este modelo es estocástico, esto es, se utilizan tres diferentes parámetros probabilísticos que controlan el proceso: p probabilidad de que cresta un árbol (transición $0 \to 1$), f probabilidad de que un árbol se incendie espontáneamente $(1 \to 2)$ y g probabilidad de que un árbol sea inmune al fuego. Las reglas son las siguientes:

1. Un sitio 0 se vuelve 1 con probabilidad p.
2. Un sitio 1 se vuelve 2 con probabilidad $(1-g)$ si al menos un vecino está en estado 2.
3. Un sitio 1 se vuelve 2 con probabilidad $f * (1-g)$ si no hay vecinos en estado 2.
4. Un sitio 2 se vuelve 0 al siguiente paso temporal.

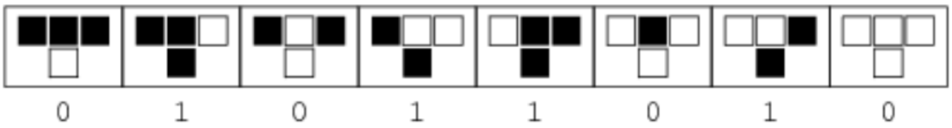

Figura 24.6 Regla 90 de Stephen Wolfram. El renglón superior muestra el estado de la célula i y sus dos vecinas al tiempo t. Abajo se muestra el valor de i en $t + 1$.

Opción 4: Reglas de Stephen Wolfram

Wolfram introduce una serie de reglas para autómatas unidimensionales, en ellos sólo se tiene un arreglo de celdas, y los posibles vecinos de una celda sólo son el que se encuentra a su izquierda y a su derecha. Cada celda sólo puede tener dos posibles estados "1" (célula viva), "0" (célula muerta). Su **regla 90** está dada por las transiciones entre estados ilustradas en la Figura 24.6. La cual se interpreta de la siguiente manera:

- Para el primer 0 lo que nos indica es que si una celda está viva y sus dos vecinos también están vivos la celda pasa a morir.
- Para el primer 1: Si una celda está viva, su vecino de la izquierda también está vivo pero su vecino de la derecha está muerto, entonces la celda permanece viva.

Y así sucesivamente se construyen las 6 reglas que faltan.

Debido a que este es un autómata unidimensional y nuestra representación gráfica es para autómatas bidimensionales, lo que haremos es lo siguiente:

- Pintaremos solo una celda viva en la mitad del primer arreglo.
- Cuando evolucione dejaremos el primer renglón tal cual y dibujaremos su evolución en el segundo renglón, y así sucesivamente.
- Cuando la evolución llegue al último renglón disponible de la malla, ocuparemos el renglón 0 para poner la evolución del autómata, y se podrá volver a empezar.

En la iteración 15 tu autómata se tendrá que ver como la Figura 24.7.

Generalidades

El proyecto debe implementar, al menos, dos de los modelos planteados aquí, se darán máximo dos puntos extra a quien implemente los cuatro correctamente.

Entradas

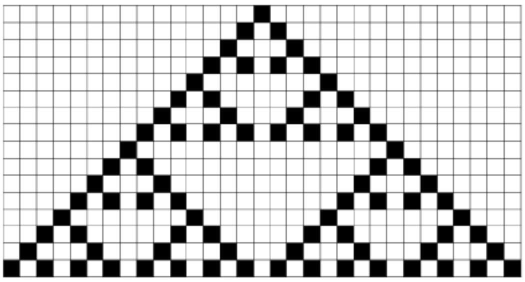

Figura 24.7 Representación bidimensional de la evolución de la regla 90. El eje vertical es el tiempo, de arriba hacia abajo.

- El tamaño de la malla.
- Parámetros de control para el autómata (umbrales, valores de cambio de estado, etc.)

Salida

- El desplegado gráfico de la evolución del autómata.

Ejecuta ahora el código muestra, si aún no lo has hecho, y observa bien cómo funciona. Luego lee todas las indicaciones siguientes antes de comenzar a programar.

1. La clase `Autómata` deberá implementar las cosas que hay que hacer en todos los tipos de autómata. Puedes comenzar programando sólo un autómata en la clase hija `AutómataUno`[1], asegúrate de que funcione bien, tratando de no modificar la clase `Autómata`.

   Cuando intentes programar el siguiente modelo, en otra clase que también extienda `Autómata`, te darás cuenta del código que tienen en común y podrás pasarlo a la clase `Autómata`. Al agregar al segundo modelo, necesitarás modificar el enum `TipoAutómata` dentro de la clase `Menús`, para que se agregue el menú correspondiente a la interfaz. En este punto, también es recomendable hacer que `Autómata` sea abstracta y arreglar el menú para que ya no se cree el autómata que no hace nada.

2. Cada subclase de `Autómata` deberá agregar, en su nodo `root`, controles que permitan al usuario asignar valores a los parámetros de cada modelo. Haciéndolo correctamente estos controles aparecerán y desaparecerán cuando el usario cambie el tipo de autómata a desplegar en el menú de la aplicación, de modo que los controles para leer parámetros que son específicos del modelo sólo aparezcan cuando se está viendo ese modelo.

[1] Es recomendable que le cambies el nombre por uno que refleje al autómata que hayas elegido programar, por ejemplo `AutómataIncendio`. Tendrás que hacer las modificaciones correspondientes al enum `TipoAutómata` en la clase `Menús`.

Para hacer esto observa cómo se agregaron los menús, botones y etiquetas en la clase `Menús`. Puedes usar las mismas ideas para agregar componentes a la raíz de cada autómata. Puedes cambiar el tipo del atributo `root` y utilizar otros objetos tipo `Parent` en su lugar. Consulta la documentación y elige lo que te resulte más conveniente. TIP: los objetos tipo `Parent` pueden contener otros objetos tipo `Parent` adentro de ellos. Recuerda que siempre necesitarás un área principal dónde dibujar al autómata, puede ser un `Group`.

3. Utiliza *eventos* para leer los datos que introduzca el usuario. Cada tipo de control tiene su forma de comunicarse con el usario. Revisa la documentación para ver qué opciones tienes.

   Una vez más, la clase `Menús` contiene ejemplos de cómo asignar objetos tipo `EventHandler` con el código a ejecutar. La notación utilizada son *lambdas* de `Java`, que implementan la interfaz funcional `EventHandler<ActionEvent>`.

4. Una vez que logres leer los parámetros propuestos por el usuario, procede a dibujar la malla de células. Te servirán los objetos de tipos que heredan de `Shape`, `Rectangle` en particular. Una sugerencia es que comiences probando a dibujarla en un método auxiliar, que puedes invocar desde el constructor de `Autómata`.

5. Desde el constructor de la subclase en la que estés trabajando, crea el estado inicial del autómata según el modelo que estés implementando, guárdalo en un arreglo de números, que deberá ser un atributo de tu objeto. Luego asegúrate que las células queden pintadas según el valor que tengan en este arreglo.

6. Ahora modifica el método `handle` e implementa ahí el algoritmo para obtener el estado del autómata en el siguiente paso temporal. Necesitarás otro arreglo para guardar el estado al tiempo $t + 1$. Actualiza los colores de las células y el atributo con los valores del autómata, para que ahora apunte al arreglo nuevo.

7. Si todo está correcto, se presentará gráficamente la evolución temporal del autómata. Las células en diferente estado deben distinguirse por su color. Los colores se dejan a elección del programador.

Referencias

Bak, P. y K. Chen (1991). «Self-Organized Criticality». En: *Scientific American* 264. Una de las fuentes primigenias del tema de la criticalidad auto-organizada. En la hemeroteca de la Facultad está la colección de *Scientific American* desde hace unos 50 años a la fecha (nota del 2003)., págs. 46-53.

Gaylord, Richar J. y Kazume Nishidate (1996). «Cellular Automata Simulations with Mathematica». En: *Modelling Nature*. La colocación en la biblioteca es QA267.5,C45,G39. Incluye muchos ejemplos más de autómatas celulares como modelos de fenómenos naturales. TELOS-Springer Verlag.

Nakanishi, Hiizu (jun. de 1990). «Cellular-automaton model of earthquakes with deterministic dynamics». En: *Physical Review A* 41.12. Una generalización del modelo presentado aquí. Además se muestra que los resultados arrojados por éste son consistentes con la ley de Gutenberg-Richter, el modelo continuo más robusto en terremotos., págs. 7086-7089.

Schönfisch, Birgitt (1995). «Propagation of Fronts in Cellular Automata». En: *Physica D: Nonlinear Phenomena* 80. Incluye un par de modelos de propagación de epidemias., págs. 433-450.