31

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES

CAMPUS ACATLAN

CAMEX

UN PROGRAMA PARA CONTROLAR MAQUINAS DE AUTOMATAS CELULARES

T E S I S

QUE PARA OBTENER EL TITULO DE:

LICENCIADO EN MATEMATICAS APLICADAS Y COMPUTACION

P R E S E N T A N :

LUCIANO DE LA ROSA AGUILAR

VICTOR MANUEL JIMENEZ ARELLANO

ASESORES DE TESIS: MTRO. JOSE MANUEL GOMEZ SOTO

DR. SERGIO V. CHAPA VERGARA

ACATLAN, ESTADO DE MEXICO NOVIEMBRE DEL 2000

CAMEX

Un Programa para controlar Máquinas de Autómatas Celulares

Luciano de la Rosa Aguilar
Victor Manuel Jiménez Arellano

A mis padres

Luciano de la Rosa Arellano y Graciela Aguilar Álvarez

Por su infinitamente amplio e ininterrumpido apoyo

A mi hermana

Ariadne de la Rosa Aguilar

Por el cariño que siempre me ha brindado

Luciano

A Gracia, Toño y Aurorita

Por todo el apoyo y cariño recibidos

Víctor

Prefacio

Este trabajo representa un esfuerzo por presentar un documento introductorio sobre el funcionamiento del programa CAMEX, y pueda ser usado como referencia para quien quiera profundizar en aspectos particulares de la CAM-PC o de CAMEX. Este texto pretende proporcionar la información suficiente para cualquier usuario que desee manejar una CAM-PC mediante CAMEX, manejando los recursos principales de la CAM-PC para la simulación de autómatas celulares.

Esperamos que este trabajo pueda ser de utilidad, ayudando a cubrir la ausencia de material introductorio sobre CAMEX, y que sirva como referencia para futuros estudios y trabajos.

LOS AUTORES

Contenido

Introducción

En la mitología griega, la maquinaria del universo eran los mismos dioses. Ellos personalmente tiraban del sol por el cielo, mandaban la lluvia y el trueno, y alimentaban las mentes humanas de pensamientos apropiados. En concepciones más recientes, el universo es creado completo con su mecanismo operativo: una vez puesto en movimiento, corre por sí mismo. Dios se sienta fuera de él y puede deleitarse observándolo.

Los autómatas celulares son universos sintéticos estilizados definidos por reglas simples como aquellas de un juego de mesa.

Toffoli & Margolus

El hombre durante mucho tiempo se ha valido de las matemáticas para representar muchos de los fenómenos que acontecen a su alrededor. Recientemente ha habido un auge por el estudio de los sistemas dinámicos que, aún rigiéndose por leyes completamente deterministas, tienen comportamientos impredecibles e inesperados.

Una de las principales razones de esta renovada curiosidad por algunos sistemas dinámicos es sin duda el advenimiento de las computadoras electrónicas, que con su extrema rapidez para realizar cálculos aritméticos han permitido conocer y experimentar cosas a las que no se podría haber tenido acceso ni aún con el trabajo de incontables generaciones de hombres.

Una clase de modelos matemáticos, que en fechas recientes ha llamado la atención de estudiosos de diversas disciplinas, son los *autómatas celulares*; modelos atractivos por ser sumamente simples, y para los que se han encontrado aplicaciones en matemáticas, física, química, biología y computación.

Algunos de los problemas que se estudian con autómatas celulares han sido estudiados tradicionalmente utilizando modelos continuos —generalmente basados en ecuaciones diferenciales parciales— y ahora se brinda una

herramienta alternativa para el modelaje y estudio de dichos fenómenos, ya que los autómatas celulares son simples y con mecanismos de operación completamente conocidos.

Una de las dificultades principales en el estudio de los autómatas celulares es que involucran una cantidad muy grande de cálculo, y se necesita mucho poder de cómputo para poder realizar los cálculos con la rapidez que nos gustaría. Mientras tanto, en espera de que una revolución tecnológica aumente considerablemente la cantidad de datos con los que pueden lidiar las computadoras de propósito general, se han construido máquinas especializadas en el estudio de estos modelos conocidas como *Máquinas de Autómatas Celulares* (CAM por sus siglas en inglés).

La CAM es una computadora diseñada en el Instituto de Tecnología de Massachusetts (MIT) que permite hacer cálculos con autómatas celulares a una gran velocidad, y a un precio moderado, lo que la hizo disponible para prácticamente toda la comunidad científica.

La CAM se controla mediante la paquetería que se distribuye con la misma máquina, pero también es posible programar un nuevo *software* para controlarla; tal es el caso de CAMEX (*CAM Exerciser*). CAMEX es un programa desarrollado por Harold V. McIntosh en el Departamento de Aplicación de Microcomputadoras del Instituto de Ciencias de la Universidad Autónoma de Puebla. Este programa tiene ciertas ventajas sobre el *software* original de la CAM, y un enorme potencial para crecer y desarrollarse.

CAMEX es un programa de distribucion gratuita, lo que lo hace accesible para cualquier persona y puede obtenerse a través de la red mundial Internet. Pero la característica que lo hace atractivo y que le otorga el mencionado potencial, es que también se distribuye gratuitamente el programa fuente con el que CAMEX fue creado, y así, al ser de código abierto, brinda a cualquier persona o institución la libertad de modificarlo. Esto es, pueden reprogramar CAMEX para hacerlo más amigable, más grande, o incluso adaptarlo para trabajar según sus necesidades particulares. Es de esta manera que CAMEX tiene posibilidades de crecer y optimizarse siendo modificado, posiblemente, de forma conjunta entre varios programadores.

El programa ejecutable y fuente de CAMEX puede obtenerse en la siguiente dirección (en donde también estar disponible la versión modificada por los autores):

`http://delta.cs.cinvestav.mx/~mcintosh/oldweb/software.html`

Existen dos CAM disponibles en la Sección de Computación del Centro de

Investigación y Estudios Avanzados (CINVESTAV) del Instituto Politécnico Nacional (IPN) y una en el Departamento de Aplicación de Microcomputadoras del Instituto de Ciencias de la Universidad Autónoma de Puebla (UAP).

El presente trabajo, tiene como objetivo principal dar a conocer CAMEX, y servir a la vez como documento de referencia para manipular y aprovechar el programa. Para mantener la uniformidad con los programas fuente originales, los comentarios del código intoducido por los autores se deja en inglés. La versión de máquinas CAM que controla CAMEX es, específicamente, la CAM-PC.

Estructura de este trabajo

En el Capítulo 1 se introduce al lector al tema de los autómatas celulares. No se profundiza en la teoría ni en el tratamiento matemáticos de los mismos, ya que ese no es el objetivo de esta tesis. Se pretende que los modelos sean comprendidos de una manera general, y se tratan principalmente conceptos que aparecerán en capítulos posteriores.

El Capítulo 2 es referente a la computadora de propósito específico CAM. Se habla de su arquitectura y funcionamiento, y de manera muy breve, se muestra como manipularla con el *software* original que se distribuye con la máquina.

Los capítulos 1 y 2 son suficientes —y necesarios— para llegar a la sección donde finalmente se trata de forma directa el tema de este trabajo. El Capítulo 3 está dedicado al programa CAMEX; a su entorno y a la manera de utilizarlo.

El diseño de experimentos a través de CAMEX es rico y variado, pero la verdadera ventaja se tiene cuando se reprograma este paquete reescribiendo los archivos fuente y volviéndolos a compilar. La manera de hacer esto es explicada en el Capítulo 4, que representa la parte medular de este trabajo. Es también en este capítulo donde se profundiza en la arquitectura de la CAM, pues un mayor conocimiento de la tarjeta es necesario para entender como funciona el programa que controla la CAM.

Para poder reprogramar CAMEX y trabajar con un programa modificado por el usuario, es necesario recompilar los archivos fuente, que están codificados en lenguaje C. Por cualquier discrepancia que pudiera producirse por el uso de distintos compiladores, en el Apéndice A se encuentra la información sobre cómo fue compilado originalmente CAMEX, mencionando la versión y nombre del compilador. Se incluyen también indicaciones para que el pro-

grama se compile sin problemas (sin advertencias ni mensajes de error) y el modelo de memoria a utilizar.

Objetivo

Presentar al programa CAMEX como una opción en el control de una Máquina de Autómatas Celulares (CAM). Desarrollar nuevos operadores con reglas y configuraciones iniciales, así como proporcionar un trabajo que cubra la deficiente documentación existente sobre el programa CAMEX, explicando el funcionamiento y manejo del programa a través de la interfaz que se proporciona al usuario, la programación de experimentos con autómatas celulares con el lenguaje de programación REC, y proveer de la información necesaria para reescribir el código de CAMEX y modificar las rutinas que controlan los recursos de la CAM encargados de manejar los principales elementos de los que constan los autómatas celulares, que son las configuraciones iniciales, los estados, las vecindades y las reglas de transición.

Agradecimientos

Queremos agradecer a todas las personas que de alguna manera ayudaron en la elaboración de este trabajo.

Agradecemos a Sergio V. Chapa Vergara y a José Manuel Gómez Soto por todas las facilidades otorgadas durante nuestra estancia en el IX Verano de la Investigación Científica en el Centro de Aplicación de Microcomputadoras de la UAP. Particularmente queremos mostrar nuestro agradecimiento a J.M. Gómez Soto por la orientación, apoyo y disposición otorgados a lo largo de la elaboración de este trabajo. A Sofi, secretaria de la Sección de Computación del CINVESTAV, por su ayuda, y a Pedro Hernández por la búsqueda de los archivos de CAMEX.

Agradecemos también a Noé Sierra y a David Cruz R. por la ayuda en lo referente a programación en C, y a todos los muchachos del Verano del 99 en Puebla (Juan Carlos Seck, Abdiel, César, Nancy y Tannia) por la convivencia. Es de mencionarse también la accesibilidad que mostró Norman Margolus, del MIT, en las consultas por correo electrónico.

Queremos mostrar nuestra gratitud para con el personal administrativo de la ENEP Acatlán que con suma amabilidad y disposición hizo posible que este trámite fuera más ágil. Ellos son la profesora Ma. Carmen González

Videgaray, la profesora Beatriz Trueba, Fernando García Minquini, Cristian Carlos Delgado Elizondo y la Lic. Dolores Aguilera y Gómez.

Por su comprensión y apoyo agradecemos a los ingenieros Carlos Valdés G., Domingo Castañeda C. y J. Francisco Piñón R., de Luz y Fuerza del Centro (Luciano), así como a los ingenieros Daniel González y Roberto Rivera (Victor).

Finalmente, queremos expresar el más especial y sincero de los agradecimientos al Dr. Harold V. McIntosh, con quien aprendimos mucho dentro y fuera de lo académico, por los agradables momentos que pasamos en su compañía.

Sobre este documento

Este trabajo fue elaborado con el sistema de preparación de documentos LaTeX 2ε y compilado con el programa MiKTeX 1.20a.

Capítulo 1

Autómatas celulares

En este capítulo se introduce al lector al tema de los autómatas celulares. Se pretende que los modelos sean comprendidos de una manera general, y se tratan principalmente conceptos que aparecerán en capítulos posteriores. Inicialmente se habla de los conceptos fundamentales de los autómatas celulares mencionándose los elementos de los que constan. La exposición sobre los autómatas celulares se hace conforme al desarrollo histórico que como campo teórico han tenido, explicándose su origen y algunos de los más significativos estudios que sobre ellos se han venido realizando.

1.1 ¿Qué son los autómatas celulares?

Los autómatas celulares son modelos matemáticos tan sencillos que pareciera como si se tratara de algún juego de mesa. A pesar de esto, son capaces de simular cosas tan complejas como una computadora de propósito general. Los autómatas celulares, por fines didácticos, casi siempre son considerados como un juego en los textos introductorios o de divulgación, y creemos que esa es la mejor forma de introducirse a ellos. Por eso comenzaremos ilustrando lo que es un autómata celular al igual que se hace en [17], haciendo como si fuéramos a realizar una animación con la ayuda de un bloc cuadriculado.

Lo primero que tenemos que hacer es rellenar con tinta negra algunos de los cuadrados que se encuentren por el centro de la última hoja del bloc, que será el primer cuadro de nuestra animación. Sobrepondremos la siguiente hoja sobre la primera, con tal de que la figura que dibujamos sea visible a través del papel. Para dibujar el cuadro número 2 de nuestra animación

sobre esta hoja, seguiremos la siguiente receta:

1. Escoge un cuadrado, y fíjate en el área de 3 x 3, conocida como vecindad (Figura 1.1), que está centrada en él. Si alrededor del cuadrado hay exactamente tres cuadrados coloreados, márcalo tenuemente con lápiz. Marca también el cuadrado si éste se encuentra sobre un cuadrado coloreado.

2. Haz lo mismo con todos los cuadrados, y cuando termines ilumina con tinta negra los cuadrados marcados.

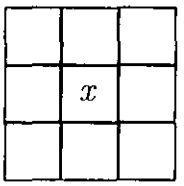


Figura 1.1: Área de 3 x 3 centrada en el cuadrado x. Se conoce como vecindad de Moore.

El cuadro 3 de nuestra animación se construirá en base al cuadro 2, así como construimos el 2 a partir del 1; el cuadro 4 se construye en base al 3; y así sucesivamente, hasta agotar todas las hojas del bloc. Como podrá imaginarse, para ver nuestra "película", ahora tendremos que sostener la libreta en la mano, y hacer resbalar una a una cada hoja por nuestro pulgar. Podrá observarse una mancha negra que crece de manera irregular, abarcando cada vez más espacio. Su contorno mostrará distintas formas y algunos cuadrados permanecerán en blanco. Algunas de las formas que podemos obtener se muestran en la Figura 1.2.

Si cambiamos la configuración inicial de cuadrados iluminados, se desarrollará una historia diferente; si cambiamos la receta, obtendremos otra dinámica. Podemos considerar otra vecindad, utilizar más colores de tinta, o incluso cambiar la malla para obtener hexágonos o triángulos (Figura 1.3) en vez de cuadrados, o trabajar con un número de dimensión diferente.

Un autómata celular es, en esencia, algo como lo que, con términos puramente ilustrativos, acabamos de describir.

Otro sencillo ejemplo de lo que es un autómata celular lo da el modelo de Greenberg-Hastings[20], que se muestra en la Figura 1.5. En este autómata

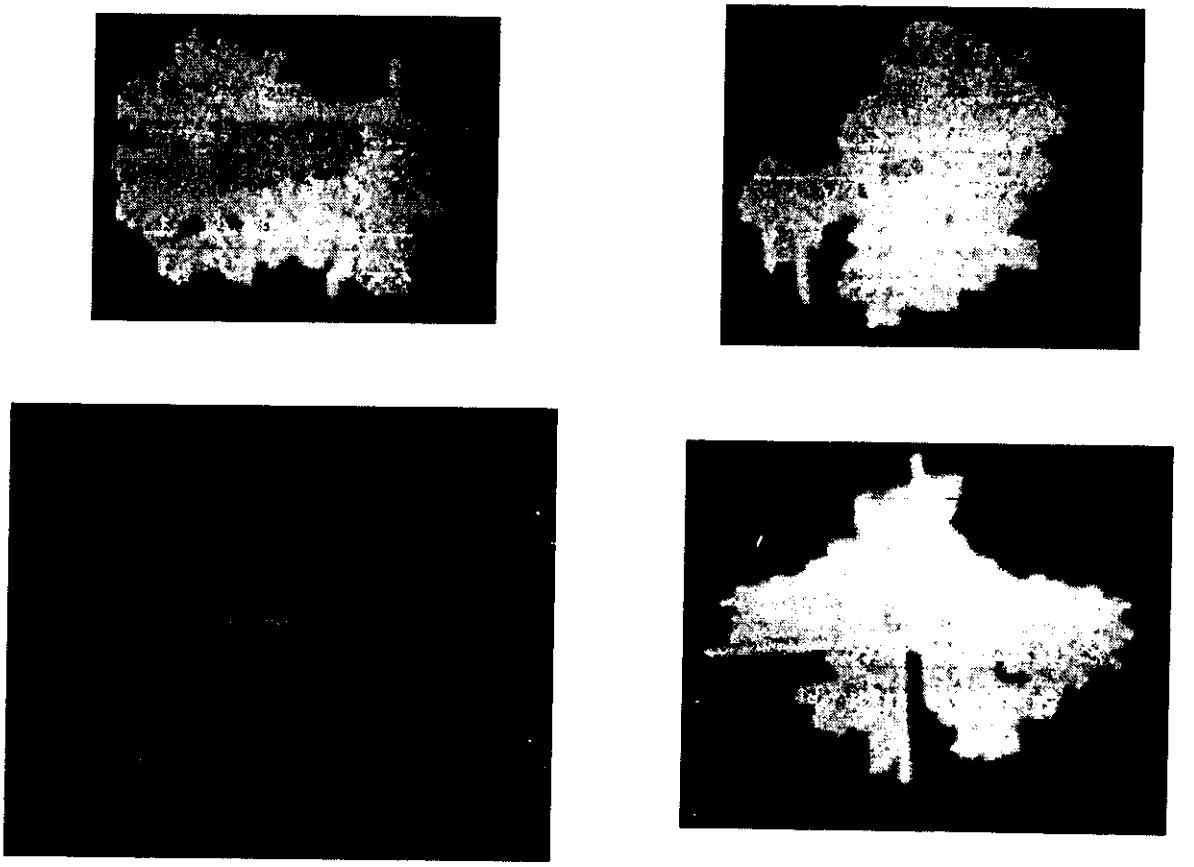


Figura 1.2: Diferentes evoluciones de la "mancha de tinta", partiendo de distintas configuraciones iniciales

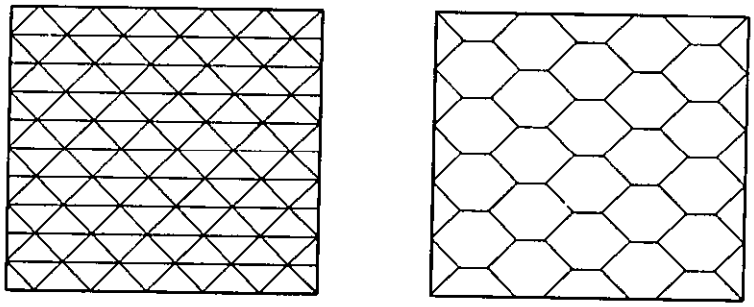


Figura 1.3: Una malla triangular y una hexagonal.

celular cada célula puede encontrarse en tres estados (colores): en reposo, excitada, y recobrándose. La vecindad considera sólamente a la vecina superior, la inferior, y las de dos lados (4 células vecinas como en la Figura 1.4). Por último la receta para este autómata celular dice que si una célula está en

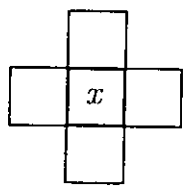


Figura 1.4: Vecindad considerada en el autómata celular Greenberg-Hastings, que consta de 5 vecinos si incluimos a la célula evaluada x. Es conocida como vecindad de von Neumann.

reposo, así permanecerá a menos que tenga por lo menos una vecina excitada, en cuyo caso pasará al estado de exitación en la siguiente generación; una célula excitada se encontrará recobrándose en el siguiente instante; y una célula recobrándose, estará en reposo en el siguiente instante. Nuevamente podemos modificar la configuración inicial y observar diferentes evoluciones del fenómeno.

Las condiciones iniciales —configuración de celdas con la que empezamos a correr nuestro autómata—, en algunos casos, determinan la evolución del

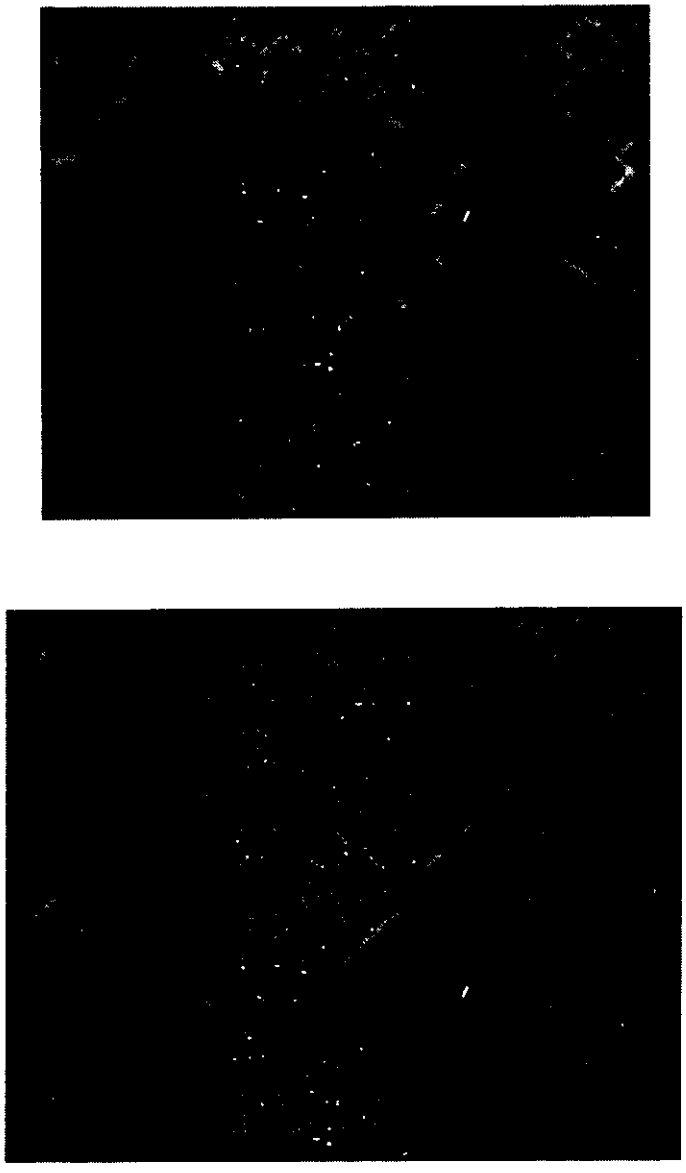


Figura 1.5: Dos evoluciones distintas generadas por la regla de Greenberg-Hastings, partiendo de configuraciones iniciales diferentes

autómata. Estas condiciones iniciales pueden ser generadas aleatoriamente o de manera arbitraria, de hecho algo muy común es observar el desarrollo de un autómata celular a partir de *semillas*, que son configuraciones iniciales donde sólo una o unas pocas células juntas están iluminadas. También es muy frecuente el observar la evolución de un autómata celular a partir de una configuración inicial donde los estados de las células se eligen aleatoriamente; a tal configuracin inicial a veces se le llama *caldo primordial.*

La elección de la regla es la que en realidad hace que el comportamiento de un autómata celular sea interesante, y frecuentemente una pequeña modificación en la regla desemboca en un drástico cambio de comportamiento. Es posible crear reglas probabilísticas donde la celda, para ser actualizada, depende de un valor tomado de una distribución de probabilidad.

En términos generales, un autómata celular tiene las siguientes características:

espacio Está formado por una rejilla o malla de celdas, (la cuadrícula de la libreta en los ejemplos anteriores).

tiempo La evolución toma lugar en intervalos de tiempo discretos.

estados Cada celda o célula se encuentra, en algún momento determinado, en un estado tomado de un conjunto finito de estados. En nuestro primer ejemplo cada celda podía tener dos estados, coloreado o en blanco. Se conviene en denominar a los estados asignándoles números enteros comenzando por el 0.

reglas de transición Cada célula evoluciona de acuerdo a una regla (receta) uniforme que depende sólo del estado actual de la célula y de los estados de un número finito de células vecinas. Uniforme significa que la regla es la misma para cualquier célula del espacio.

vecindad La vecindad o vecindario es uniforme al igual que la regla, y también es local, lo que significa que sólo se consideran células cercanas.

Los autómatas celulares simulan satisfactoriamente algunos fenómenos naturales en donde se lleven a cabo interacciones entre partículas cuyos cálculos sean paralelos, locales, y homogéneos.

1.1.1 Historia de los autómatas celulares

La teoría de los autómatas celulares comenzó a mediados de la década de los cincuentas cuando el matemático nacionalizado estadounidense John von Neumann se propuso probar que era posible construir máquinas que se pudieran autorreproducir. Visualicemos lo siguiente: una máquina está rodeada por materiales de construcción y está construyendo otra máquina siguiendo los planes que tiene almacenados en su propia memoria. Como el plan fue almacenado para su construcción, la máquina que construye tendrá el mismo plan en su memoria. También será capaz de construir una máquina, y así puede seguir la cadena por siempre. Debido a las dificultades evidentes de construir materialmente una máquina con tales características, Stanislaw M. Ulam sugirió a von Neumann llevar a cabo su objetivo de una manera abstracta.

La nueva prueba de von Neumann usó lo que ahora llamamos "espacio celular uniforme", equivalente a un tablero de ajedrez infinito. Cada célula (o celda) tiene un número finito de *estados* y un conjunto finito de células *vecinas* que pueden influenciar su estado. El patrón de los estados cambia en pasos discretos de tiempo de acuerdo a un conjunto de reglas de transición que se aplican simultáneamente a cada célula. Podríamos decir (como en [14]) que von Neumann puso el "autómata", y S. Ulam lo "celular". Cada celda puede considerarse como un autómata, y el autómata celular está formado por muchas celdas todas iguales.

Von Neumann[18], aplicando reglas de transición a un espacio en el que cada célula tenía 29 estados y cuatro células ortogonalmente adyacentes, como en la Figura 1.4, probó la existencia de una configuración de unas 200,000 células que se autorreproducirían. La razón para tan enorme configuración es que, para la demostración de von Neumann, era necesario que su espacio celular fuera capaz de simular una máquina de Turing, es decir, un autómata, llamado así por su inventor, el matemático británico Alan M. Turing, capaz de llevar a cabo cualquier cálculo deseado. Metiendo esta computadora universal en su configuración, a von Neumann le fue posible producir un constructor universal evitando el crecimiento de cristal —que no puede considerarse estrictamente como autorreproducción—, obteniendo cierta variabilidad, adaptabilidad y evolución[10]. Aparte el nuevo modelo seguramente podría ser tratado matemáticamente, y podrían establecerse teoremas como los que Turing encontró para la computación. Como el modelo de von Neumann podría, en principio, construir cualquier configuración

deseada en una región vacía del espacio celular, podría autorreplicarse cuando se le diera un esquema de sí mismo. A partir de su muerte en 1957, su prueba de existencia ha sido enormemente simplificada. Edgar F. Codd[1] volvió más simple el modelo de von Neumann. En realidad la historia de los autómatas celulares puede rastrearse aún a tiempos anteriores a los del trabajo de von Neumann [9], como por ejemplo los estudios sobre redes neuronales de Warren S. McCulloch y Walter Pitts en 1943, o las ideas de Norbert Wiener que con la aparición de su libro en 1948 fundaron el inicio de la ciencia cibernética. En estas obras anteriores a las máquinas autorreproductivas de von Neumann no existía el concepto de autómata celular como tal, pero ya contenían algunos elementos que seguramente inspiraron el desarrollo de lo que ahora conocemos como autómatas celulares.

1.2 El Juego de la Vida

A pesar de ser mucho más accesible que el de von Neumann, el diseño de Codd [3] tampoco pudo ser puesto en acción, pues manejaba 8 estados con 5 vecinos y eso aún representaba una dificultad para implementarlo en las computadoras de la época. Sin embargo, se convirtió en el antecedente para el autómata celular mundialmente conocido como el *Juego de la Vida*[2], o simplemente *Vida* (*The Game of Life* o *Life* por su nombre en inglés), desarrollado por el matemático de Cambridge John Horton Conway; pues se había visto que era posible lograr el objetivo original de von Neumann pero con un modelo mucho más sencillo (Codd redujo el número de estados de 29 a 8) que podría simplificarse todavía más. El autómata celular de Conway, como veremos, era mucho más sencillo que el de Codd.

En Vida, cada célula tiene dos estados y puede estar "viva" o "muerta," utilizando el vocabulario de Conway para los estados 0 y 1; cada célula en el espacio tiene ocho células vecinas, cuatro adyacentes ortogonalmente y cuatro diagonalmente (Figura 1.1); y las reglas de Conway son las siguientes:

- *Supervivencia.* Cada célula viva con 2 o 3 vecinos vivos, sobrevive para la siguiente generación.

[1] Mejor conocido por haber sido el inventor de las bases de datos relacionales.

[2] A través de la red Internet, pueden obtenerse de forma gratuita dos excelentes programas del Juego de la Vida que corren bajo Windows: *Life32* y *WinLife*. *Life32* puede obtenerse en http://psoup.math.wisc.edu/Life32.html, mientras que en ftp://ftp.digital.com/pub/games/winlife.zip se puede conseguir *WinLife*.

- *Muerte.* Cada célula viva con 4 o más vecinos vivos "muere" (su estado cambia de 1 a 0) por sobrepoblación. Cada célula viva con 1 o 0 vecinos vivos, muere por "desolación".

- *Nacimiento.* Cada célula muerta adyacente a exactamente 3 vecinos vivos —no más, no menos— es un "nacimiento", y estará viva en la siguiente generación.

Hay que recordar que todos los nacimientos y muertes ocurren simultáneamente.

La popularidad del juego de Conway comenzó cuando Martin Gardner[5][3] escribió sobre él en su columna mensual sobre pasatiempos matemáticos en la revista Scientific American, y creció tanto que se convirtió casi en un culto. De ahí en adelante una avalancha de nuevos documentos vendrían vertiginosamente. Robert T. Wainwright mantuvo[4] durante 3 años un periódico trimestral donde informaba de los últimos descubrimientos sobre Vida. La combinación balanceada entre reglas y estados que hizo Conway había llevado a las más sorprendentes series de artefactos —cosas que parpadean, que oscilan, que se deslizan, que arrojan deslizadores, y que comen deslizadores— culminando en la demostración de que Vida era capaz de hacer computación universal, es decir, que podía construirse una computadora de propósito general con este autómata celular[5]. Muchos resultados interesantes fueron obtenidos por el Laboratorio de Inteligencia Artificial del MIT con las facilidades gráficas de su computadora PDP-6. Décadas más tarde, con la popularización de las microcomputadoras, revistas como Byte, Omni, y la misma Scientific American revivieron el entusiasmo por el juego de Conway.

[3]Este artículo fue recopilado junto con otros más en [6]

[4]Y en cierta forma sigue manteniendo, ahora por medio de su página personal en Internet: http://members.aol.com/lifeline/life/lifepage.htm. En realidad no es una revista eletrónica, sino que en esta página se encuentran los primeros números del boletín, información general sobre Vida, y se presenta cada mes un patrón (configuración inicial) distinto.

[5]A muy grandes rasgos, para hacer computación con Vida lo que se hace es lanzar deslizadores (*gliders*) unos contra otros utilizando patrones (configuraciones) conocidos como pistolas de deslizadores (*glider guns*). Cuando los deslizadores chocan entre sí, pueden ocurrir cosas como el que desaparezcan o continúen su camino. Si los deslizadores pueden pasar asumimos que es un 1 y si no, que es un 0. Puede probarse que se puede realizar la construcción de las compuertas lógicas fundamentales en las que se basan las computadoras. John H. Conway detalló cómo Vida podía realizar computación universal en el segundo volumen de un libro titulado "Winning Ways"[1], del que fue coautor. Puede obtenerse un poco de información al respecto en el siguiente sitio:
http://alife.santafe.edu/alife/topics/cas/ca-faq/lifefaq/lifefaq.html

Vida y sus predecesores creados por von Neumann y Codd proveyeron un testimonio concreto de que matemáticas arbitrariamente complicadas podían ser reproducidas dentro de un sistema cuya organización básica era completamente rudimentaria.

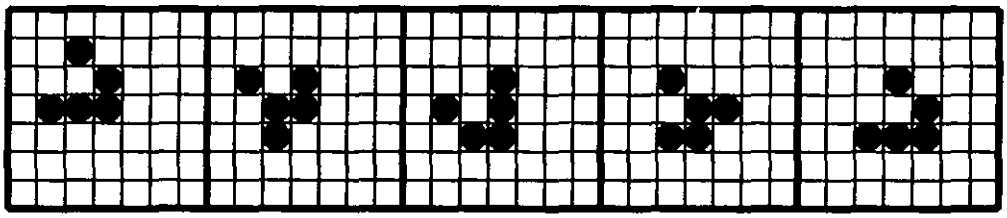


Figura 1.6: Uno de los famosos *deslizadores* del Juego de la Vida, formado por cinco células vivas. Después de cuatro generaciones aplicando las reglas de Conway la figura se desplaza por el espacio un cuadro hacia la derecha y uno hacia abajo, volviendo a su forma original.

1.3 La clasificación de Wolfram

Según Harold V. McIntosh (en [9]), pueden considerarse tres etapas fundamentales en la historia de los autómatas celulares:

- La era von Neumann
- La era Gardner
- La era Wolfram

De estos tres puntos, el primero corresponde a John von Neumann y sus máquinas autorreproductivas, el segundo a Martin Gardner, que popularizó el juego de Conway y, el tercero, a Stephen Wolfram por la clasificación que hizo de los autómatas.

Durante las décadas de los cincuentas, sesentas y setentas hubo una cantidad tremenda de investigación sobre autómatas, lenguajes y tópicos similares, pero los autómatas celulares no recibían mucha atención y se le debe a John Conway y a su juego el que los autómatas celulares se hayan vuelto

populares. Puede decirse por todo esto que la "era Gardner" se caracterizó por la búsqueda intensa de configuraciones "interesantes".

El interés científico en los autómatas celulares recibió un ímpetu con las investigaciones de Stephen Wolfram[21], quién a principios de los ochentas analizó sistemáticamente autómatas de una dimensión. Wolfram, en contraste con von Neumann y Conway, que se concentraron en sólo una regla que servía a sus propósitos, analizó un gran número de reglas registrando el desarrollo del autómata celulara largo plazo y comparando las historias evolutivas. Wolfram pudo realizar estos estudios gracias a la capacidad que las computadoras de la época ya habían alcanzado.

Wolfram propuso las siguientes cuatro clases de autómatas celulares:

Clase I. Comprendía a los autómatas que evolucionaban -generalmente muy rápido— hacia un campo de quietud. (Ver Figura 1.7).

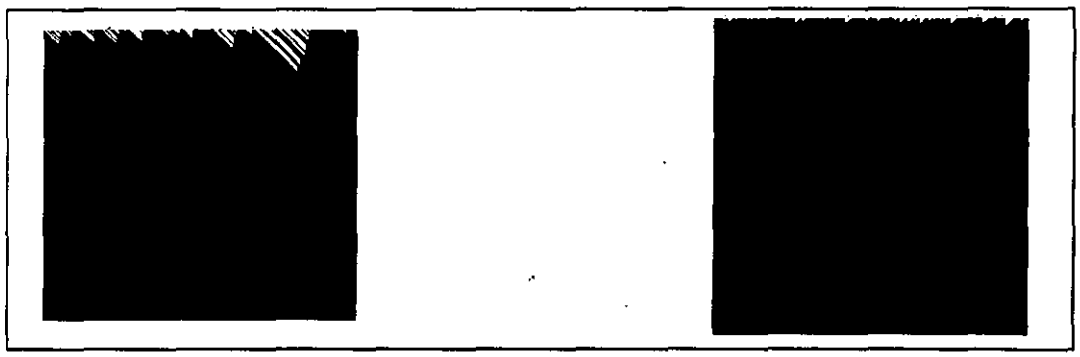


Figura 1.7: Dos reglas de autómatas (2,1) que pueden clasificarse como pertenecientes a la clase I

Clase II. Son los autómatas con conducta cíclica. (Ver Figura 1.8).

Clase III. Es la completamente opuesta a la anterior e incluye comportamientos caóticos impredecibles. Ni local ni globalmente se encuentran patrones periódicos reconocibles (Ver Figura 1.9).

Clase IV. Esta última clase es la más interesante y la menos común. Chris Langton sugiere que esta clase está entre la segunda y la tercera porque, consiste en "islas de comportamiento caótico en un mar de calma"[9]. (Ver Figura 1.10).

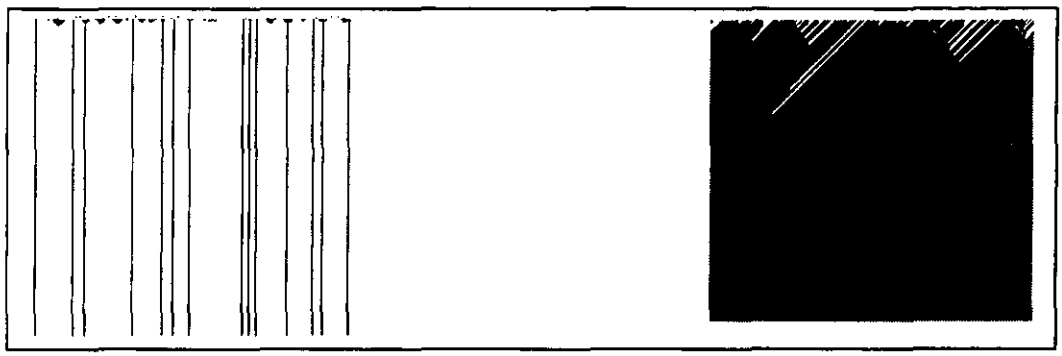


Figura 1.8: Dos reglas de autómatas (2,1) que pueden clasificarse como pertenecientes a la clase II

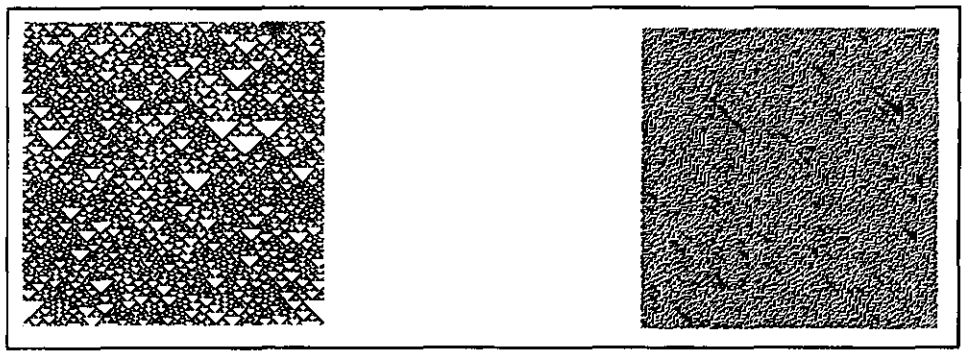


Figura 1.9: Dos reglas de autómatas (2,1) que pueden clasificarse como pertenecientes a la clase III

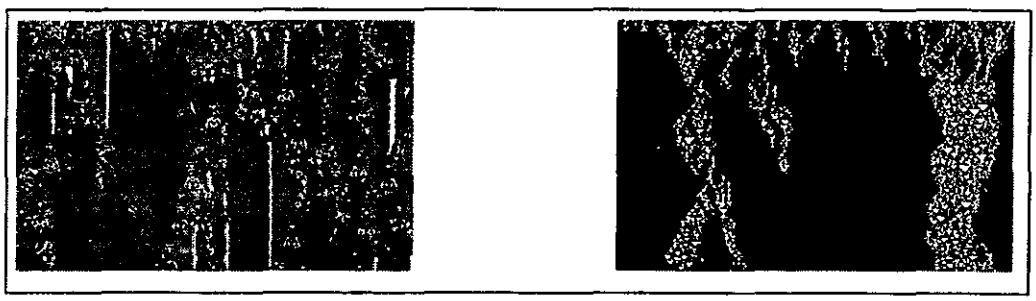


Figura 1.10: Dos reglas de autómatas (4,1) que pueden clasificarse como pertenecientes a la clase IV

Al juzgar las clasificaciones que hizo Wolfram, parece haber sido inspirado por la teoría dinámica de sistemas y su posible relación con la teoría de los autómatas celulares. De cualquier forma hay que tener muy claro que su clasificación es completamente empírica, lo que en otras palabras significa que no existe ningún método cuantitativo o analítico para decidir si un autómata celular pertenece o no a alguna clase.

1.4 Autómatas celulares unidimensionales

Para hacer un autómata celularunidimensional se colocan las células en línea una detrás de otra como en la Figura 1.11. Todas las células tienen el mismo número de estados y las reglas de transición son las mismas en cada una de las celdas. En cuanto a la vecindad, lo más común es relacionar cada una de las células con un determinado número de celdas vecinas a cada lado. Una línea finita de celdas consecutivas debe tener necesariamente dos extremos. Las celdas en los extremos pueden ser tratadas de manera diferente o, para evitar el considerar "casos especiales", podemos unir la célula incial con la célula final, haciendo que nuestro arreglo de células forme un anillo (Fig. 1.12), en donde la primera célda será el vecino derecho de la última, y la última el vecino izquierdo de la primera. Este es un recurso ampliamente utilizado y el que más sirve a nuestros propósitos, ya que no se rompe la uniformidad del autómata celular, aunque para la simulación de ciertos fenómenos físicos podría ser conveniente otro tratamiento de bordes[16, 20].

1.4.1 Notación matemática

Los autómatas celulares pueden definirse con la precisión necesaria para que se les pueda aplicar un análisis matemático, y para esto debemos contar con una notación adecuada. Stephen Wolfram ha propuesto una notación matemática para autómatas celulares que es ampliamente utilizada.

Sea r el número de vecinos a cada lado de una célula (lo que dará una vecindad simétrica). De esta forma la vecindad completa estará conformada por $2r + 1$ células (r células de cada lado, más la del centro).

En notación de Wolfram escribiremos (k, r) para denotar un autómata celularen el que cada célula posee k estados y r vecinos a cada lado.

Una sencilla convención para definir una regla de transición específica para autómatas $(2, 1)$, consiste en asignarle a cada regla un número deci-

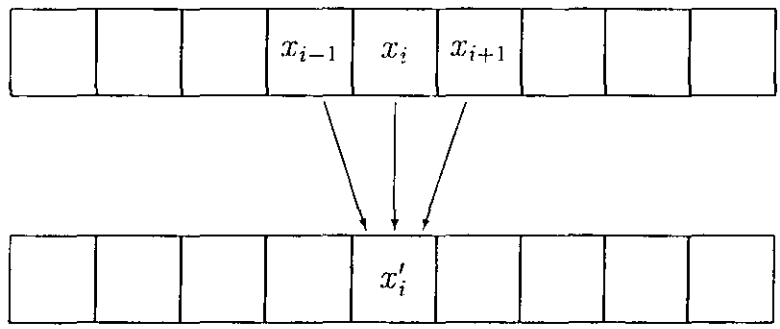


Figura 1.11: Se muestra una vecindad de tres vecinos, donde la célula evaluada x_i tiene un vecino a cada lado. El mapeo al aplicar una regla se realiza sobre la celda x_i', que ocupa el mismo lugar de x_i en el arreglo actualizado, que se coloca en el siguiente renglón.

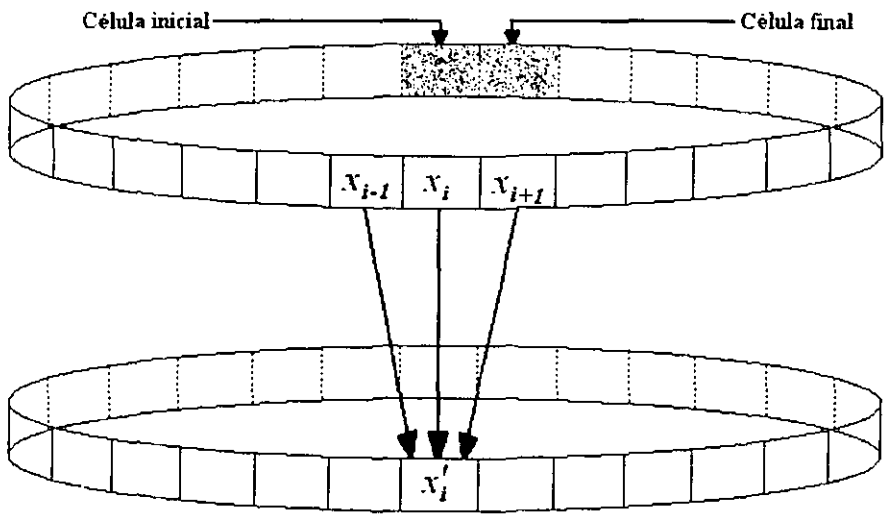


Figura 1.12: Las celdas de los bordes se unen formando un anillo.

mal. Para ilustrar este procedimiento, obsérvense las siguientes secuencias de números:

111	110	101	100	011	010	001	000
0	0	0	1	0	1	1	0

El primer renglón muestra todos los casos de vecindades posibles cuando un autómata celular unidimensional tiene un vecino a cada lado ($r = 1$), y el segundo renglón es una regla específica, que indica hacia que estado se evoluciona con determinada vecindad. Las vecindades ilustradas en el primer renglón están ordenadas "de mayor a menor" (si las consideramos números) y el número decimal con el que determinaremos esta regla será la conversión del número binario de ocho dígitos que aparece en el segundo renglón, siendo en este caso el 22. Decimos entonces que ésta es la regla decimal 22 de Wolfram.

Resultan evidentes las ventajas de utilizar una notación matemática común en vez de tablas para ilustrar las funciones que definen el siguiente estado de una celda, así es que será ϕ el símbolo que utilizaremos para denotar la función de $2r + 1$ argumentos que define una transformación local de la vecindad hacia un nuevo estado. Así, tenemos la siguiente expresión,

$$x_i^{t+1} = \phi(x_{i-r}^t, x_{i-r+1}^t, \ldots, x_{i-1}^t, x_i^t, x_{i+1}^t, \ldots, x_{i+r-1}^t, x_{i+r}^t)$$

donde x_i^t es el valor —tomado del conjunto de estados—, de la célula en el sitio i en el tiempo t, y ϕ es una función arbitraria que especifica la regla.

Viendo la notación matemática anteriormente expuesta, se ve que para un autómata celular en, por ejemplo, dos dimensiones, cada variable x tendría dos subíndices i y j para indicar la posición de la célula por renglón y columna.

Erica Jen[8] utiliza una práctica fórmula para obtener el número decimal de una determinada regla. Por ejemplo, para los autómatas celulares (2,1), se denotarán las ocho vecindades posibles como $000 \to a_0, 001 \to a_1, \ldots, 111 \to a_7$, donde $i = 0, 1, \ldots, 7$ y $a_i \in \{0,1\}$. De esta manera, para obtener el número decimal de una regla basta con aplicar la siguiente fórmula

$$R = \sum_{i=0}^{7} a_i 2^i$$

En el caso más general, donde tuviéramos k estados, podríamos usar la misma fórmula sustituyendo el 2 por el número k.

1.4.2 Reglas totalísticas

Mediante un sencillo análisis de combinatoria, se obtiene la fórmula $k^{k^{2r+1}}$ para el número de autómatas (k, r) existentes. Observamos entonces cómo el aumento exponencial hace que el análisis exhaustivo de ciertos tipos de autómatas se torne imposible con el poder de cómputo actual (por esta razón también se utilizan conjuntos de estados pequeños[6]). Para obtener una muestra manejable de entre todas las reglas posibles, Wolfram ideó lo que se conoce como *reglas totalísticas*, haciendo que la transición dependa sólo del valor de la suma de los números asignados para representar los estados. La regla de Vida es, de hecho, una regla totalística, pues el estado siguiente de una célula depende de la suma de los valores o pesos de sus vecinas y de sí misma[7].

La regla 22 mencionada anteriormente, es también una regla totalística. La suma de tres dígitos binarios (por la vecindad que incluye tres células cada una con dos estados posibles) pueden sumar 0, 1, 2 o 3. Arreglando las sumas de mayor a menor e indicando el estado correspondiente a la evolución de la suma, podemos obtener un número decimal para representar la regla totalística (el 2 en este caso), de la misma manera que se obtiene un número decimal para especificar una regla particular.

suma	3	2	1	0
nuevo estado	0	0	1	0

1.4.2.1 Notación matemática de reglas totalísticas

Al depender sólo de la suma de los estados de los vecinos, la función matemática de una regla totalística para un autómata celular$(k, 1)$ sería

$$\phi(a, b, c) = \phi(a + b + c)$$

donde ϕ es una función entera de argumentos enteros y en módulo k.

[6]Una razón para utilizar un número grande de estados sería el aproximar un autómata celular a un sistema continuo. Esto podría realizarse haciendo que cada célula tuviera un número de estados lo suficientemente grande como para representar un número real[10].

[7]Con más precisión, la regla de Vida es *semitotalística*, ya que —a diferencia de la regla 22— para llevar a cabo la transición se debe considerar antes si la célula evaluada está viva o muerta. Existe también la discusión de si una regla totalística debe considerar en la suma el valor de la célula analizada. Se llama regla totalística externa a la que sólo contempla a las células vecinas en la suma.

1.4.3 Vecindades fraccionarias

Podríamos pensar que la vecindad más pequeña es la de los autómatas $(k, 1)$, donde la vecindad consta de 3 celdas teniendo cada celda un vecino a cada lado. Pero la verdad es que podemos tener vecindades fraccionarias y construir, digamos, un autómata celular$(k, \frac{1}{2})$, donde $2r + 1 = 2$. Para hacer esto, se remplaza la vecindad completa (de dos celdas) por una sóla celda, como en la Figura 1.13.

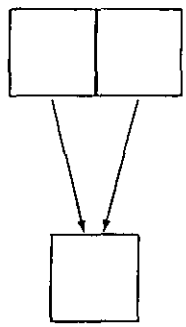


Figura 1.13: Mapeo de una vecindad de $2r + 1 = 2$ vecinos donde $r = \frac{1}{2}$.

En general, r puede ser cualquier número no entero siempre y cuando $2r + 1$ sí sea un número entero; para que sea un número entero de celdas el que mapea en un nuevo estado.

1.5 Autómatas celulares en dos dimensiones

La expansión a dos dimensiones es significativa para la representación de varios fenómenos físicos. Ya hemos visto a principios del capítulo un par de ejemplos de autómatas celulares en dos dimensiones, y también hemos explicado el autómata celularbidimensional Vida, en donde teóricamente la cuadrícula se extiende sobre un área infinita. Ante esta imposibilidad práctica, se conviene en hacer algo similar al anillo que se forma con los autómatas unidimensionales. Para tratar los bordes extremos en un autómata celularbidimensional, el borde izquierdo se "pega" al derecho, y lo mismo se hace con los bordes superior e inferior. De esta forma la superfcie que antes era un plano, tendrá ahora la topología de un toro (en términos no matemáticos esta figura sería la de una dona), ya que no es posible formar

una esfera con una malla regular de tamaño arbitrario. Para formar un toro se unen el renglón superior con el inferior y la columna de la izquierda con la de la derecha. Así, un objeto —como un deslizador de Vida— que se saliera del borde de la pantalla, entraría en la misma posición pero por el borde opuesto, tal como sucede en algunos videojuegos, como el popular *Pac-Man*.

A diferencia de los autómatas celulares unidimensionales, donde los vecinos únicamente pueden ubicarse a los lados de la célula en cuestión, en los autómatas celulares bidimensionales tenemos varias alternativas para elegir. Para empezar, la rejilla de un autómata celular en dos dimensiones no tiene por qué ser necesariamente cuadriculada, aparte de cuadrados hay dos posibles rejillas regulares, que pueden estar formadas por triángulos o por hexágonos. En el caso de una malla triangular cada celda tendrá tres vecinos, y para una malla hexagonal habrá seis células vecinas (Figura 1.3). En este trabajo trataremos únicamente con células cuadradas.

La vecindad ortogonal que consta de 5 celdas (contando la célda evaluada) como la del modelo de Grennberg-Hastings es conocida como vecindad de von Neumann (Fig. 1.4), y la vecindad que consta en total de 9 celdas —la vecindad de von Neumann más las celdas diagonales, como en Vida—, es conocida como la vecindad de Moore (Figura 1.1). Estas dos vecindades pueden generalizarse para un cierto radio (formándose cuadros y diamantes (rombos) de tamaño arbitrario), y en general puede definirse cualquier vecindad arbitraria[8], siempre y cuando ésta sea finita y se mantenga la uniformidad.

Esta sección tratará sobre autómatas celulares en dos dimensiones un poco más a fondo de lo que ya se ha hablado sobre ellos, aunque esencialmente el tratamiento es el mismo que para los autómatas celulares de una dimensión, y prácticamente lo único que se hace es "extrapolar" los conceptos a dos dimensiones.

Para observar la evolución de los autómatas celulares unidimensionales, utilizábamos renglones. En los autómatas celulares bidimensionales usaremos planos. De esta forma, tendremos nuestra configuración inicial en un plano cuadriculado en donde cada una de las celdas puede estar en uno de varios estados posibles. Después de aplicar la regla de evolución a todas y cada una de las células que componen nuestro plano, obtendremos otra configuración, que representaremos en otro plano distinto, e iremos viendo sucesivamente

[8]Se puede observar que una vecindad de Moore o de von Neumann de radio suficientemente grande puede contener a cualquier vecindad arbitraria y por esto tiene una ventaja teórica, pero en la práctica, una vecindad grande implica utilizar más recursos de *hardware* o *software*.

plano tras plano cómo nuestro autómata celularva cambiando. En la Figura 1.14 se ilustra la transición que se da de un plano a otro en los autómatas celulares bidimensionales utilizando una vecindad de von Neumann, y en la Figura 1.15 utilizando la vecindad de Moore.

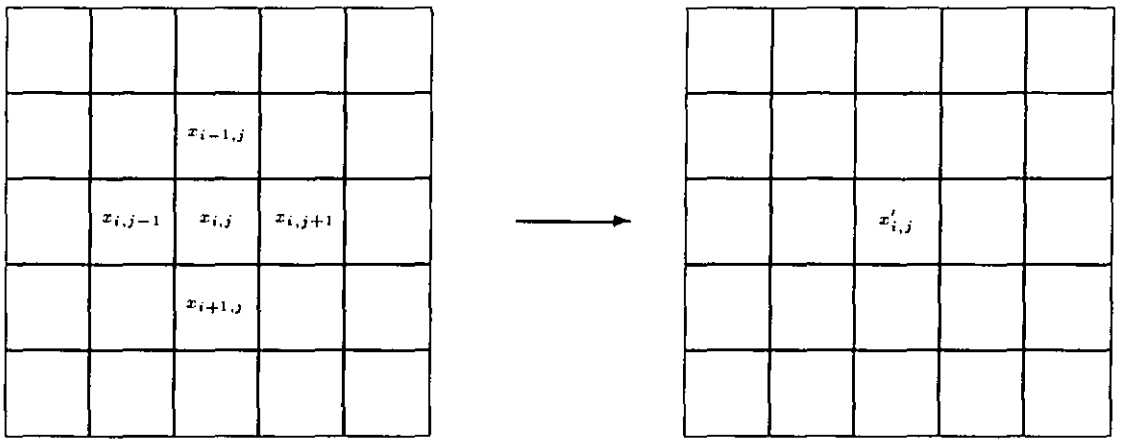


Figura 1.14: $x_{i,j}$ se transforma en $x'_{i,j}$.

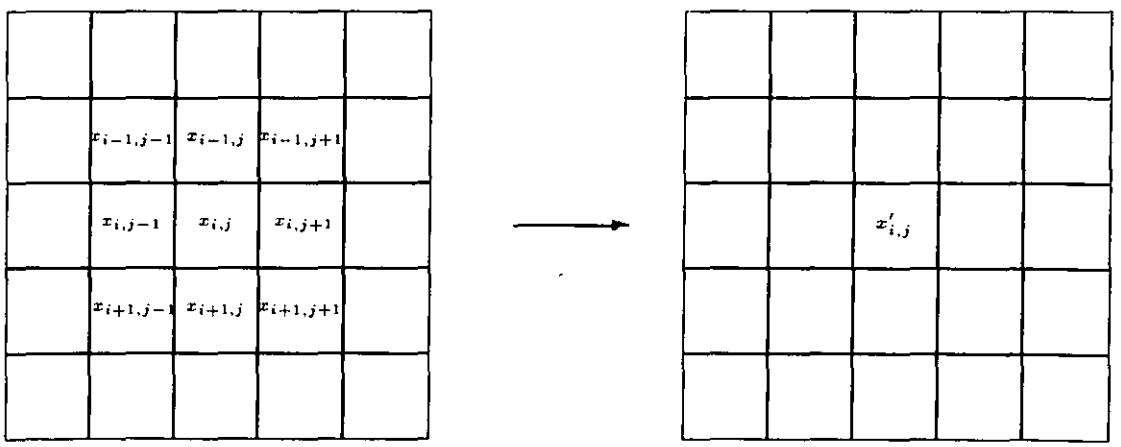


Figura 1.15: $x_{i,j}$ se transforma en $x'_{i,j}$.

1.5.1 Notación matemática

Después de haber visto la notación matemática de los autómatas celulares de una dimensión, notaremos como en dos dimensiones se hace algo muy similar, con la única diferencia de ahora, como ya no nos encontramos en una línea, sino en un plano, tendremos que considerar a las células vecinas que se encuentran arriba y abajo de la célula evaluada, y no sólo a los lados. La ecuación que describe la función de transición de un autómata celular bidimensional tiene la particularidad de que cada símbolo que represente a una célula deber tener dos subíndices (i y j, como en una matriz) para distinguir a las celdas que se encuentren en un plano de dos dimensiones.

La función que define la transición en un autómata celular bidimensional podemos representarla como

$$x_{i,j}^{t+1} = \phi(x_{i-r,j-r}^{t}, \ldots, x_{i,j}^{t}, \ldots, x_{i+r,j+r}^{t})$$

donde x_i^t es el valor —tomado del conjunto de estados—, de la célula en el sitio i en el tiempo t, y ϕ es una función arbitraria que especifica la regla.

1.5.2 Reglas totalísticas

Hemos visto como, al ir agregando estados o vecinos a un autómata celular, los cálculos crecen exponencialmente, y en dos dimensiones este es un problema mucho mayor que en los autómatas celulares de una dimensión. En un autómata celular como el del Juego de la Vida, donde manejamos dos estados y nueve células conformando la vecindad, tenemos un total de 2^{512} (2^{2^9}) posibilidades distintas para asignar una regla. Es por esto que en los autómatas celulares de dos dimensiones también tiene grandes ventajas el utilizar reglas totalísticas.

1.5.2.1 Notación matemática de reglas totalísticas

La notación matemática de reglas totalísticas se hace, igual que en los autómatas celulares unidimensionales, haciendo que el nuevo estado de la célula evaluada dependa sólo de de una suma de argumentos, y no de alguna regla arbitraria que contemple cómo están las celdas distribuidas espacialmente.

Nuestra ecuación para un autómata celular de dos dimensiones utilizando una regla totalística sería:

$$x_{i,j}^{t+1} = \phi(x_{i-r,j-r}^{t} + \cdots + x_{i,j}^{t} + \cdots x_{i+r,j+r}^{t})$$

Podemos considerar al Juego de la Vida como una regla semitotalística, porque en Vida la transición no depende solamente de la simple suma de sus vecinos. Es decir, en Vida no tenemos una regla como que diga algo así como "si la suma es X el nuevo estado será 0; y si no, 1", sino que en Vida, hay un factor adicional que hay que considerar, y éste es el estado actual en que se encuentra la célula evaluada. De esta manera, la regla de Vida es como sigue:

- Si $x_{i,j}^t = 0$ y $\phi(x_{i-1,j-1}^t + \cdots + x_{i+1,j+1}^t) = 3$ entonces $x_{i,j}^{t+1} = 1$
- Si $x_{i,j}^t = 1$ y $\phi(x_{i-1,j-1}^t + \cdots + x_{i+1,j+1}^t) > 3$ o < 2 entonces $x_{i,j}^{t+1} = 0$
- $x_{i,j}^t$ permanece en su estado (1 o 0) en cualquier otro caso.

En general, para definir una regla semitotalística para un autómata celular de dos estados, debemos preguntarnos lo siguiente: Si el estado de la celda evaluada es 0, ¿con qué suma cambiará a 1 y con qué suma permanecerá en 0? Y si el estado es 1, ¿con qué suma cambiará a cero y con qué suma permanecerá en 1?

Experimentando con reglas totalísticas y semitotalísticas podemos obtener una muestra mucho más manejable que el número que obtendríamos si consideráramos todas las reglas posibles, y aún así tenemos un gran conjunto de reglas en donde podemos encontrar dinámicas muy interesantes. En los autómatas celulares bidimensional con $r = 1$ (un radio de vecindad igual a 1) y con dos estados, existen dinámicas más interesantes que en los autómatas (2,1) en una dimensión , ya sea que consideremos o no reglas totalísticas.

1.6 Autómatas celulares en tres dimensiones

En tres dimensiones por lo general sólo se consideran rejillas cúbicas, por la facilidad de manejarlas y representarlas. Como rejillas tridimensionales más complicadas podemos considerar aquellas que constan de secciones de cubos, por ejemplo los prismas triangulares que quedan al seccionar al cubo diagonalmente con un plano. En los autómatas celulares tridimensionales se expande la teoría de la misma forma que se expande de una a dos dimensiones, y de esta manera, las vecindades de Moore y von Neumann estarían dadas por cubos adyacentes en vez de cuadrados, como en dos dimensiones. La vecindad de Moore estaría dada por 27 cubos en un arreglo de 3x3x3, formando un

cubo más grande (ver Figura 1.16); y la vecindad de von Neumann por 7 cubos(ver Figura 1.17).

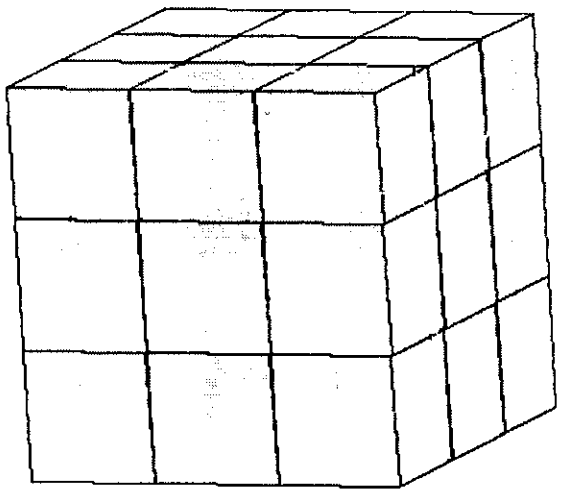


Figura 1.16: Vecindad de Moore en tres dimensiones.

A veces, la evolución de un autómata celular bidimensional es observada en tres dimensiones. Lo que se hace es ir sobreponiendo los planos respectivos a cada generación uno sobre otro, de la misma manera que observamos la evolución de una autómata celularunidimensional en dos dimensiones. Hay que diferenciar entre un autómata celularde dos dimensiones cuya evolución se observa en tres dimensiones y uno definido originalmente en tres dimensiones.

La cantidad de cálculos que necesitan realizarse para estudiar autómatas celulares en tres dimensiones es mucho mayor que los que se necesitan en una y dos dimensiones, y por esta razón, los estudios sobre autómatas celulares tridimensionales son pocos.

1.6.1 Notación matemática

Ya se ha presentado una notación matemática para autómatas celulares de una y dos dimensiones. De manera análoga, podemos representar la transición de un autómata celular tridimensional de la siguiente forma:

$$x_{i,j,k}^{t+1} = \phi(x_{i-r,j-r,k-r}^{t}, \cdots, x_{i,j,k}^{t}, \cdots, x_{i+r,j+r,k+r}^{t})$$

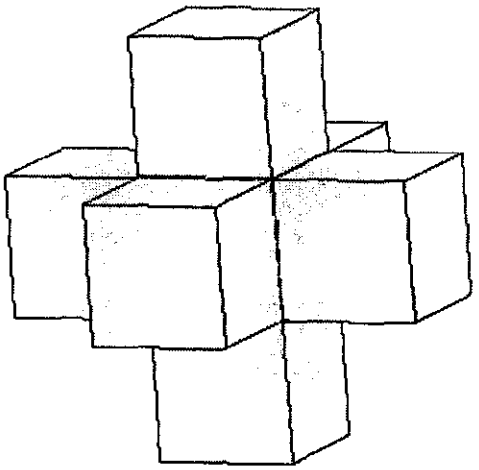


Figura 1.17: Vecindad de von Neumann en tres dimensiones.

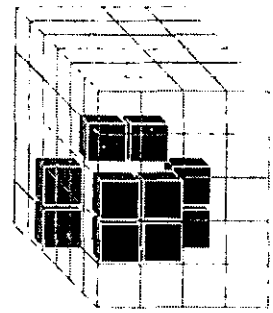
1.6.2 Vida en tres dimensiones

Carter Bays, especialista en ciencias de cómputo de la Universidad de Carolina del Sur, exploró versiones tridimensionales que pueden considerarse como las sucesoras del juego Vida original[4]. Al tratarse de tres dimensiones cada célula es, claro está, un cubo en vez de un cuadrado y la vecindad está conformada por 27 células en vez de 9. Bays llama a sus creaciones Vida 4555 y Vida 5766, bajo una denominación ideada por él mismo. Los dos primeros dígitos indican la suerte de las células vivas, el primero indica el número mínimo de vecinos que una célula ha de tener para permanecer viva, y el segundo número el máximo para que la célula no se asfixie por la sobrepoblación. Los dígitos tercero y cuarto gobiernan la suerte de las células muertas, indicando el número mínimo de vecinas que una célula muerta ha de tener para cobrar vida y, el número máximo de vecinas para hacer lo mismo, respectivamente. El juego original en dos dimensiones sería, entonces, Vida 2333.

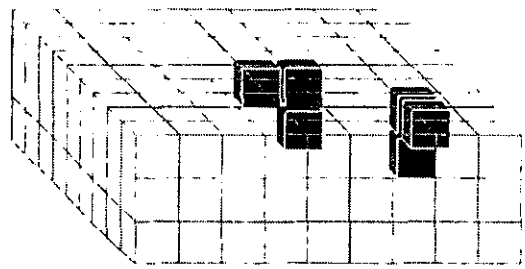
Vida 4555 funciona de manera similar a Vida 2333, (y curiosamente se obtiene sumando 2 a cada cifra del código del juego de Conway) y Bays pudo observar un deslizador, agrupaciones que no cambian y patrones que oscilan (Figura 1.18). Bays también produjo numerosas colisiones entre deslizadores y otras configuraciones, obteniendo resultados interesantes.

Vida 5766 parece asentarse (se vuelve monótona) mucho más rápido que

Ejemplo de un glider del Juego de la Vida tridimensional para la regla 4-5, 5-5...



Ejemplo de un glider del Juego de la Vida tridimensional para la regla 5-6, 5-5...



Ejemplo de un glider del Juego de la Vida tridimensional para la regla 5-7, 6-6...

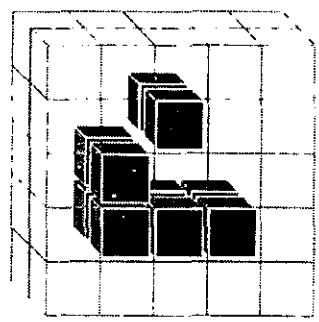


Figura 1.18: Diferentes *gliders* encontrados en reglas distintas de Vida en 3 dimensiones.

Vida 4555. Pero lo interesante de Vida 5766 es que, bajo ciertas condi
encontradas por Bays, simula varios aspectos de Vida de Conway en el pla

1.7 Otras dinámicas de evolución

En esta sección se ilustran otras reglas de transición de autómatas celulares bidimensionales, que es en los que se basa nuestro interés. Esto se hace con dos propósitos: el que el lector tenga una idea más amplia sobre el tema, brindándole más ejemplos de autómatas celulares; y que se comprenda la dinámica de las reglas que los autores añadieron al programa CAMEX.[9]

1.7.1 Cuadrados, diamantes y triángulos

Esta regla es muy sencilla; se utiliza la vecindad de Moore y consiste en hacer un "OR" no exclusivo con todas las células de la vecindad. De esta manera, si al menos una célula de la vecindad está en el estado 1, la célula evaluada será 1, y en otro caso permanecerá en el estado en el que se encontraba. Al poner un 1 en un campo de ceros, observaremos como a partir de esta semilla crece un cuadrado a una tasa uniforme, hasta cubrir por completo la pantalla. Si "plantamos" más semillas, veremo varios cuadrados surgir y traslaparse en su crecimiento. Este es un ejemplo de crecimiento sin restricciones, en el que cuando una celda alcanza el valor de 1, así permanece.

Si cambiamos la vecindad de Moore por la de von Neumann observaremos el surgimiento de diamantes en vez de cuadrados y podemos obtener triángulos rompiendo la simetría de la vecindad (es decir, no se considera la vecindad de von Neumann —que es simétrica— completa). Para obtener figuras triangulares, ignoraremos a una célula de la vecindad. Por ejemplo si con la vecindad de von Neumann no tomamos en cuenta la celda que se encuentra hasta abajo, observaremos el crecimiento de triángulos apuntando hacia abajo, y en general, obtendremos triángulos apuntando en la dirección de la célula que omitamos de la vecindad, siempre y cuando ésta no sea la del centro.

[9]Todas las reglas que se presentan en esta sección, así como el autómata celular de la "mancha que crece" y el de Greenberg-Hastings (expuestos al inicio del capítulo), se encuentran en la versión de CAMEX modificada por los autores.

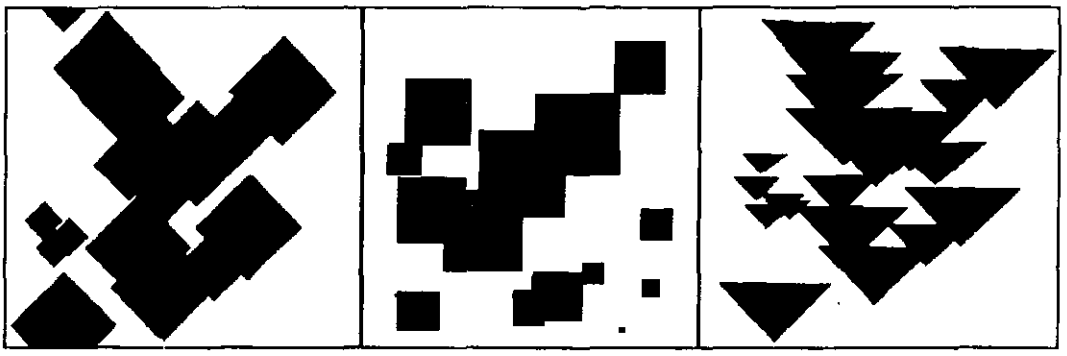


Figura 1.19: Imagen de la evolución de las reglas de diamantes, cuadrados y triángulos.

1.7.2 1 de 8

En la sección anterior vimos como un área crecía "tan rápido como podía", es decir, una célula con valor 0 tomaba el valor de 1 en cuanto tenía una vecina en estado 1. En este autómata celular limitamos este crecimiento siendo más selectivos al permitir que una célula tome el valor de 1. Con esta regla, una célula toma el valor de 1 si una y sólo una celda de sus 8 vecinas tiene el valor de 1, y permanece sin cambio de otra forma. Se observa que aquí el crecimiento es más lento que en los ejemplos anteriores y que se forma un patrón fractal regular.

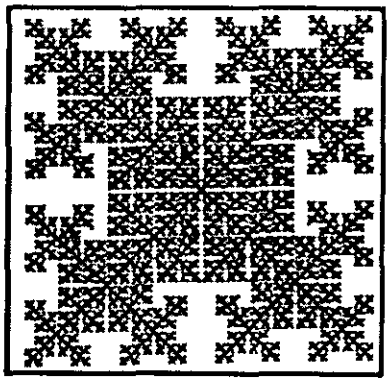


Figura 1.20: Imagen de la evolución de la regla 1 de 8.

1.7.3 Líquenes

Tenemos crecimiento con restricciones en la regla de *Líquenes*, donde una célula toma el valor de 1 si tiene exactamente 3, 7 u 8 vecinas en estado 1, y permanece sin cambio en cualquier otro caso.

Un crecimiento con competencia lo obtendremos con una variación de Líquenes, si permitimos que las células en 0 "vuelvan a vivir" a expensas de las células en 1. En esta regla, *Líquenes con muerte*, la única variación que se produce respecto a Líquenes es que la célula evaluada tomará el valor de 0 si tiene exactamente cuatro vecinas en estado 1. Los resultados de la evolución de esta regla son, en general, muy difíciles de predecir, al igual que en el Juego de la Vida.

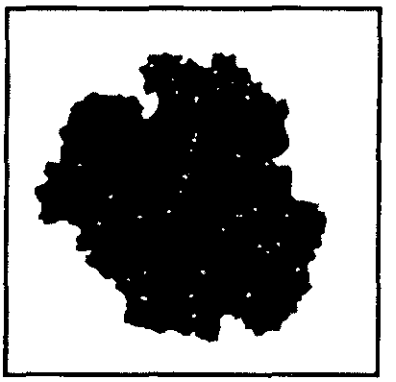


Figura 1.21: Imagen de la evolución de la regla de Líquenes.

1.7.4 Votación

En esta regla interpretamos el estado de las celdas de la vecindad como un "voto", y la célula evaluada seguirá el estado de la mayoría de los vecinos (el voto de la célula evaluada también cuenta). Así las cosas, si la suma de los votos es 5 o más, la celda tomará el valor de 1, y en otro caso el de 0. El aspecto de interés con esta regla radica en las fronteras donde coinciden áreas de unos con áreas de ceros.

Una variación interesante de la regla de Votación se obtiene haciendo que una célula con 4 vecinos en 1 pase al estado 1, y que una con 5 pase a 0.

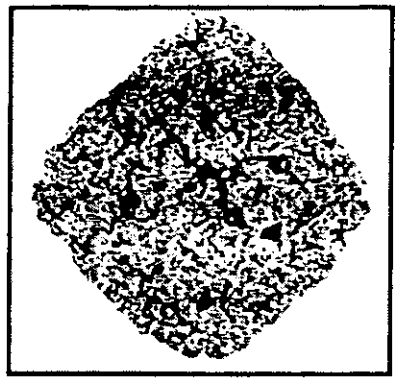


Figura 1.22: Imagen de la evolución de la regla de Líquenes con muerte.

Figura 1.23: Imagen de la evolución de la regla de Votación.

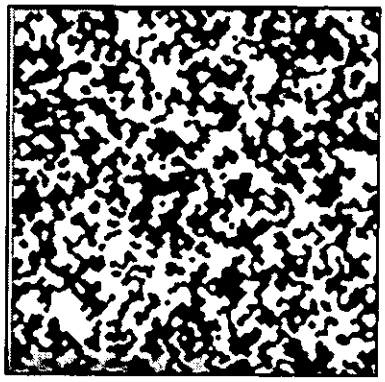


Figura 1.24: Imagen de la evolución de la regla de Votación modificada para los casos de 4 y 5 vecinos en estado 1.

1.7.5 Paridad

Esta regla, sugerida por Edward Fredkin, especifica que una célula seguirá la *paridad* de su vecindario; que estará en estado 1 o 0 dependiendo de si hay un número par o impar de células "vivas" en la vecindad. Se puede definir esta regla realizando una suma en módulo 2 con todas las celdas de la vecindad, incluyendo a la célula evaluada. En la regla de paridad se usa la vecindad de von Neumann, aunque también puede utilizarse la de Moore. Podemos representar esta regla de la siguiente manera:

$$x_{i,j}^{t+1} = x_{i,j}^{t} \oplus x_{i+1,j}^{t} \oplus x_{i-1,j}^{t} \oplus x_{i,j-1}^{t} \oplus x_{i,j+1}^{t}$$

Quizás el aspecto más llamativo de esta regla es su *linealidad*, lo que significa que si corremos dos configuraciones iniciales diferentes durante un determinado número de pasos y después sumamos (en módulo 2) las dos configuraciones resultantes, obtendremos el mismo patrón que si hubiéramos sumado las dos configuraciones iniciales y las hubiéramos corrido por el mismo número de pasos. La suma que se realiza en este caso es un OR exclusivo, es decir, una suma booleana.

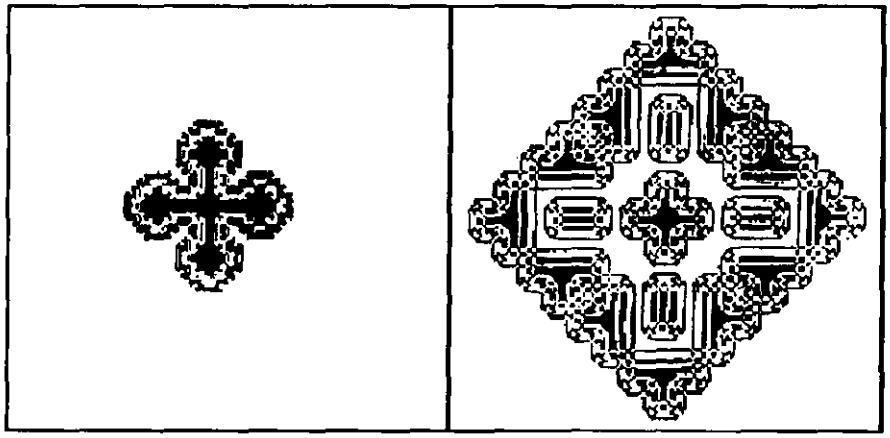


Figura 1.25: Imagen de la regla de Paridad en distintos de pasos de evolución.

1.7.6 Borde/Hueco

La regla de este autómata celular es mucho muy diferente a todos los que hemos mencionado anteriormente, se rige por una de las llamadas *reglas*

compuestas, lo que significa que distintas reglas tomarán lugar en diferentes pasos en el tiempo. La regla de transición de *Borde/Hueco* es la siguiente:

- En pasos pares, se forma un borde de unos alrededor de cualquier célula que esté en estado 1.
- En pasos impares, se "ahuecan" las áreas sólidas al tomar el valor de 0 cualquier célula que esté completamente rodeada de unos.

Este autómata celular utiliza la vecindad de Moore.

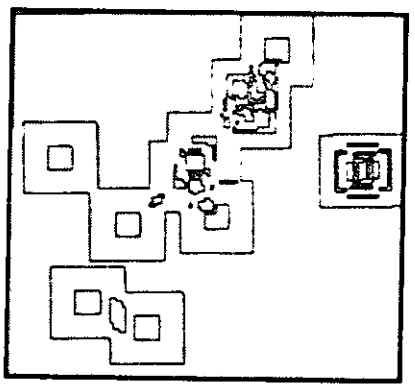


Figura 1.26: Imagen de la evolución de la regla de Borde/Hueco después de unos cuantos pasos de evolución.

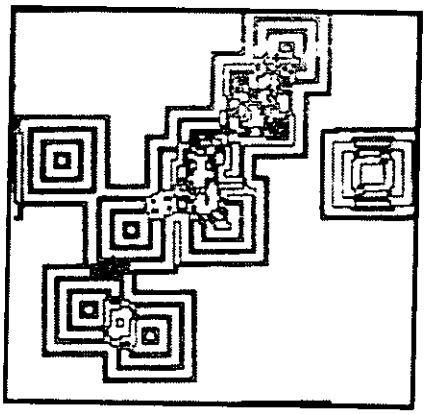


Figura 1.27: Otra etapa de la evolución de la regla Borde/Hueco.

1.8 Algo más sobre autómatas celulares

Se ha dado en este capítulo una introducción a los autómatas celulares. No obstante, a casi medio siglo de su surgimiento hay muchas facetas que se han explorado y que ni siquiera se tocan en este texto. Los autómatas celulares simulan una cantidad considerable de fenómenos físicos como solitones (donde una estructura pasa a través de otras sin deformarse), crecimiento dendrítico, percolación, dinámica de fluidos; y son utilizados para estudiar reversibilidad de fenómenos físicos. Simulan reacciones químicas no lineales como la de Belousov-Zhabotinsky; y en biología tienen aplicaciones en simulaciones a nivel celular o poblacional. Los autómatas celulares son también usados para realizar modelos de tráfico y de actividad eléctrica neuronal. La teoría de sistemas dinámicos los utiliza para estudiar la autoorganización en los sistemas, y en computación se les ve como modelos de computación en paralelo. Ejemplos de todas estas aplicaciones se encuentran en [20] (tráfico), y en [17] y [7] (todas las demás).

También se han desarrollado herramientas matemáticas para estudiar estos modelos discretos y se han adaptado conceptos de matemáticas —contínuas y discretas— para estudios analíticos y estadísticos en los autómatas celulares.

El panorama es tan amplio que incluso se dan áreas de especialización dentro del mismo campo de los autómatas celulares.

La computadora de la que hablaremos en el siguiente capítulo y el programa del que hablaremos en los últimos dos capítulos conforman una herramienta que puede resultar de suma utilidad para estudiar muchas de las facetas de los autómatas celulares en las que podamos tener interés.

1.9 Comentarios finales

Creemos que el campo teórico de los autómatas celulares tiene un gran potencial para ser estudiado y aprovechado en el futuro. Esto no significa que se deban de abandonar los modelos continuos basados en ecuaciones diferenciales, pero los autómatas celulares pueden tomarse como alternativa y quizás nos deparen resultados muy interesantes dentro de algún tiempo (por no mencionar los resultados que ya han dado).

Como Toffoli y Margolus lo hacen notar, una de las causas por las cuales los modelos matemáticos son de la forma en que los conocemos, fue la limitante que el ser humano tiene para tratar con grandes volúmenes de datos; así que tuvimos que conformarnos con modelos de unas cuantas ecuaciones y unas cuantas variables, que representaran satisfactoriamente los fenómenos naturales. Pero ahora los computadores electrónicos nos brindan una nueva posibilidad para estudiar aspectos de la naturaleza desde nuevas perspectivas, como a través modelos espacialmente distribuidos y discretos tales como los autómatas celulares.

Capítulo 2

Máquinas de autómatas celulares

Todas las ventajas que los autómatas celulares poseen tienen un costo, y es que en lugar de unas cuantas variables que interactúan entre ellas mediante ciertas relaciones —como en ciertos modelos matemáticos tradicionales (como algunos sistemas de ecuaciones diferenciales)— los autómatas celulares manejan muchas variables, ya que es una variable por cada celda, y para obtener información significativa necesitamos utilizar muchas celdas, y hacer que el autómata celular corra durante largos periodos de tiempo. En este capítulo, hablaremos sobre las Máquinas de Autómatas Celulares, que como se verá salvan eficientemente algunas dificultades concernientes al estudio de los autómatas celulares.

En realidad, en el estudio de los autómatas celulares, los límites están impuestos por la capacidad computacional con la que la tecnología actual nos permite contar. El calcular un gran número de eventos puede tomarnos mucho tiempo si queremos obtener información significativa o realizar un estudio ambicioso con autómatas celulares. Es de aquí de donde nace la necesidad de una plataforma (*hardware*) con el tamaño, velocidad, y flexibilidad adecuados para la experimentación general sobre autómatas celulares, y a un costo moderado; un simulador de autómatas celulares capaz de actualizar miles de celdas en muy poco tiempo; una *máquina de autómatas celulares* (CAM por sus siglas en inglés[1]). La necesidad de esta herramienta llevó al desarrollo de la CAM en el Laboratorio para las Ciencias de la Computación del MIT.

[1] *Cellular automata machine.*

Fue una intención de antemano el que, después de satisfacer las necesidades internas del MIT, estuviera a disposición de la comunidad científica, al costo más bajo posible. Esta es la razón por la cual se decidió diseñarla como una tarjeta de circuitos para computadoras ampliamente disponibles, y con el uso de un monitor de, relativamente, bajo costo. Esta máquina, que se hizo disponible comercialmente mediante la empresa *Automatrix, Inc.* es la versión llamada CAM-PC[2]. En este texto nos referiremos indistintamente a este dispositivo como CAM-PC o simplemente como CAM.

Mientras la PC brinda soporte físico, poder, almacenamiento en disco, un monitor y un ambiente estándar, el trabajo de simular autómatas celulares a una gran velocidad es hecho por la tarjeta. Según los creadores de la tarjeta, en esta aplicación específica el desempeño de una CAM es comparable al de una CRAY-1,[3] cuyo costo era de cientos de miles de dólares.

La CAM, aparte de los rápidos cálculos, provee al usuario la habilidad de observar los resultados de los cálculos de los autómatas celulares en tiempo real, es decir, durante la computación de los mismos. La evolución de los autómatas celulares se observa en un monitor distinto al de la computadora huésped.[4]

El *software* de la CAM está escrito en *Forth* y corre en una IBM-PC con 256 K de memoria. Con este mismo lenguaje de nivel medio (Forth) se programan nuevos autómatas celulares.

Al comercializarse, la CAM-PC pudo ordenarse por correo a *Automatrix, Inc.* dentro de un paquete de *hardware/software*, o individualmente como expansión del paquete básico, ya que varios módulos pueden ser conectados simultáneamente a una PC.

2.1 Arquitectura y operación de la CAM

Físicamente, la CAM-PC es una tarjeta de circuitos que se conecta a una IBM-PC, -XT o -AT, o cualquier modelo compatible, con sistema operativo DOS.

El conocimiento de la arquitectura de la CAM-PC es necesario para utilizar

[2]La CAM-PC es una versión que se maneja prácticamente igual a una CAM-6, su predecesora inmediata, pero fue construida con una tecnología más avanzada. Ver la Sección 4.8 para una breve historia de la CAM.

[3]Supercomputadora paralela que consta de 64 procesadores.

[4]Es posible trabajar con un solo monitor pero es mucho menos práctico.

mejor sus recursos o para programadores que deseen crear su propia interfaz con el usuario. El *hardware* de la CAM está inspirado en los mismos autómatas celulares, y en apariencia trabaja como si fuera una máquina SIMD (*Single Instruction Multiple Data*) donde un flujo de instrucciones opera sobre muchos datos distintos, justo como los autómatas celulares, donde la misma regla de transición se aplica a todas las celdas en paralelo.

La CAM contiene circuitería especial y conectores para un multiproceso, y hasta 8 tarjetas pueden ser combinadas en un solo sistema.

La CAM acualiza un arreglo de 256 x 256 células 60 veces por segundo (y añadiendo tarjetas se incrementa el volumen de datos proporcionalmente). Cuando se conectan varios módulos una tarjeta funge como amo y las demás como dispositivos esclavos.

Consideremos un autómata celular bidimensional y de dos estados. Al tener cada célula dos estados podemos considerar a cada una como un *bit* (dígito binario), y al plano en el que se encuentran como un *arreglo* o *plano de bits*. Al desear un número mayor de estados, por ejemplo 4, usualmente los denominaríamos por los numerales 0, 1, 2, y 3. Pero para comprender mejor la forma en la que se almacenan los estados en una CAM, será mejor representarlos en sistema binario, y tendremos los 4 estados representados como 00, 01, 10, y 11 y nos será útil visualizar a cada pareja de bits cómo uno apilado encima del otro. Considerando el arreglo completo, habrá ahora dos planos de bits, uno encima del otro. Disponemos, en una CAM, de 4 planos de bits, con lo que cada célula puede tener hasta 16 estados.

Es conveniente considerar planos de bits a la hora de programar, pero no a la hora de visualizar el estado de una célula en la pantalla. Así que cada célula, que será un punto o pixel en la pantalla, tendrá un determinado color según el estado en que se encuentre.

Aún cuando las reglas de transición se pueden especificar usando construcciones de un lenguaje de programación de alto nivel, la CAM las convierte internamente en una tabla de reglas (que en este texto llamaremos *tablas de consulta*), donde se lista explícitamente el próximo estado de una celda según todas las combinaciones posibles de estados de sus vecinas.

En lo posterior, consideraremos únicamente el caso de autómatas celulares en dos dimensiones.

2.1.1 Componentes principales de la CAM

Los principales componentes de la CAM-PC son el *Buffer de Planos*, el *Procesador* CAM, la *Interfaz de Video* y la *Interfaz de la PC*, como se muestra en la Figura 2.1.

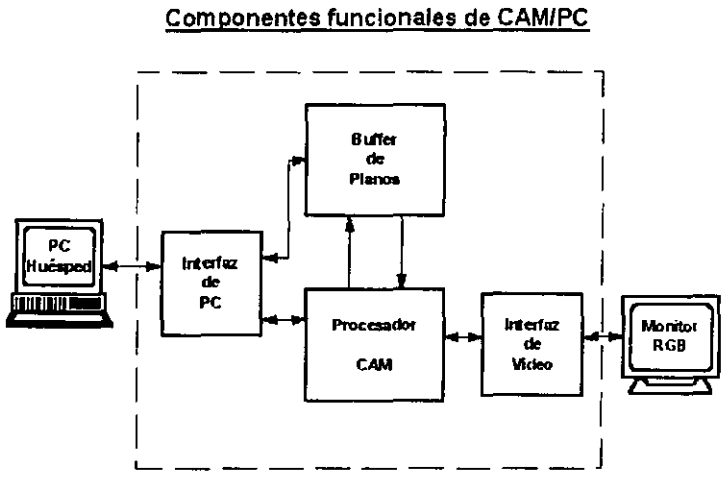


Figura 2.1: Esquema de los principales componentes de la CAM

La Interfaz de la PC es el medio mediante el cual la CAM y la computadora huésped interactúan, permitiendo que el huésped tenga acceso a los recursos de la CAM, como los registros del Buffer de Planos y del Procesador CAM. También provee a la CAM de avisar al huésped cuando ha completado sus cálculos. El computador huésped brinda un mecanismo para guardar y organizar la información, desde este punto de vista puede considerarse a la CAM como un coprocesador de la máquina huésped.

El Buffer de Planos es el principal componente de almacenamiento de la CAM, contiene los datos que serán usados por el Procesador CAM y guarda los resultados de las operaciones del Procesador CAM. La PC huésped o el Procesador CAM pueden leer el Buffer de Planos, así como escribir en él.

La Interfaz de Video provee una forma de ver la salida arrojada por el Procesador CAM como una imagen a color en un monitor de video a colores. Pero lo que es más, la Interfaz de Video permite el despliegue de los cálculos del Procesador CAM en tiempo real, como una imagen animada. La Interfaz de Video convierte el flujo de datos del Procesador CAM en una señal entendida por los monitores a color TTL y provee los niveles de voltaje eléctrico propios para controlar el monitor. El color de un pixel está determinado por el estado de la célula correspondiente y un mapa de colores que asigna un color para cada estado posible. Debido a los cuatro planos de bits, el mapa de colores tiene dieciseis entradas.

El Procesador CAM genera casi todas las señales de sincronización (*timing signals*) y controla todos los otros componentes de la CAM. También provee sincronización con otras tarjetas CAM. La función primaria del Procesador CAM es leer datos del Buffer de Planos, realizar las transformaciones de los autómatas celulares y escribir los datos otra vez en el Buffer de Planos. Estas actualizaciones pueden ocurrir en los cuatro planos de bits 60 veces por segundo. Los retrasos normales en el desplegado en el monitor, permiten a la PC huésped intercambiar parámetros con la CAM y así modificar reglas o el mapa de colores sin alentar la tasa de 60 pantallas por segundo. El Procesador CAM tiene dos procesadores independientes de transformación: CAM-A y CAM-B. Cada uno controla el procesamiento de dos planos de bits en el Buffer de Planos. CAM-A procesa los planos 0 y 1, y CAM-B los planos 2 y 3.

2.1.2 Modos de operación de la CAM

La CAM tiene tres modos principales de operación, que son el *modo ocioso*, el *modo de despliegue* y el *modo de procesamiento* o modo *paso a paso* (*stepping* en inglés). En el modo ocioso el Procesador CAM está inactivo y es usado para el intercambio de datos y valores de registro entre el huésped y la CAM. En el modo de despliegue la CAM opera como un buffer de cuadros (*frame buffer*) leyendo secuencialmente datos desde los Buffers de Planos y transmitiéndolos como imagen al monitor de video. El modo paso a paso es el modo de operación más complejo, donde los datos son leídos desde el Buffer de Planos, transformados, y escritos nuevamente en el Buffer de Planos. Los datos de la lectura o la escritura pueden ser mandados a la Interfaz de Video, de manera

que el modo de despliegue puede considerarse como subconjunto del modo de procesamiento, cuando los datos no son escritos de nuevo en el Buffer de Planos.

2.2 Vecindades y reglas

Un aspecto fundamental de los autómatas celulares es la vecindad. La CAM, al estar construida específicamente para el estudio de estos modelos, trata con los vecinos y vecindades desde el nivel del *hardware*, y debemos comprender cómo se manejan las vecindades desde el *hardware* para poder explotarlas al programar nuestros experimentos. Antes de explicar como implementar determinada vecindad en la CAM tenemos que conocer la estructura de las tablas de consulta, que son las que consideran a una vecindad.

Recordemos que para actualizarse, la CAM funciona con *tablas de consulta* (*look-up tables*), que son las tablas en donde se especifica la regla. La CAM es, virtualmente, una máquina en paralelo y podemos considerar que cada una de las células tiene su propia tabla de consulta y que todas la miran al mismo tiempo a la hora de actualizarse. Aunque en realidad lo que ocurre es que la tabla se comparte secuencialmente celda por celda, y no se aprecia el resultado sino hasta que todas las células han sido actualizadas, dando la impresión de que todas las células se actualizan simultáneamente.

Como hemos visto al analizar el Procesador CAM, éste está organizado en dos "mitades" idénticas llamadas CAM-A y CAM-B. Cada media CAM maneja dos bitplanos (planos 0 y 1, y planos 2 y 3, respectivamente) y cada mitad puede correr un autómata celular diferente, ya que cada media CAM tiene su propia tabla de búsqueda. Las dos mitades no son completamente excluyentes y cada mitad tiene un acceso limitado a la otra, como se verá más adelante.

2.2.1 Las tablas de consulta

La tabla de consulta para un autómata celular binario (de dos estados) con vecindad de von Neumann, puede ser considerada como una "caja negra" en donde, como en la Figura 2.2, se tienen cinco líneas de entrada y una de salida, donde las líneas de entrada son las cinco células de la vecindad (centro, norte, sur, este y oeste), y la línea de salida es el bit que representará el nuevo estado de la célula evaluada, que en este caso se encuentra en el Plano 0.

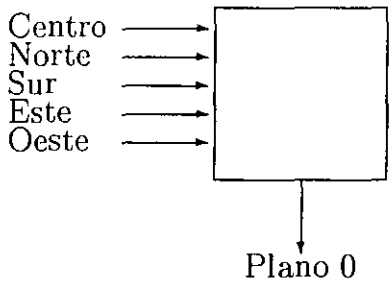


Figura 2.2: Tabla de consulta para un autómata celularbinario con vecindad de von Neumann. Las entradas son los bits de cada una de las células que conforman la vecindad, y la salida es el bit que representa al nuevo estado de la célula evaluada.

Ya que la regla de un autómata celular en una CAM se encuentra en una tabla donde explícitamente se muestran los resultados para todas las combinaciones posibles de estados en una vecindad, es necesario que exista un límite para el número de entradas en una tabla, que crece exponencialmente conforme se añaden bits (más células o estados) a la vecindad. El número de entradas en la tabla dependerá de las líneas de entrada que utilicemos, y éstas a su vez dependerán del autómata celular que queramos representar. Por ejemplo, para un autómata celularde dos estados y con vecindad de von Neumann necesitamos utilizar cinco líneas de entrada de la tabla, en donde cada línea de entrada, al tener dos valores posibles, es equivalente a cada una de las células que conforman la vecindad de von Neumann. De esta forma el número de entradas (o renglones) en la tabla será de 32 (2^5), para uno como Vida la tabla tendría 512 (2^9) entradas, y para uno con vecindad de von Neumann pero con cuatro estados por célula se tendrían $2^{10} = 1024$ entradas. El límite superior que impone la CAM es de una tabla con 16 bits de entrada, con lo que el tamaño máximo de la tabla será de 65,536 entradas (2^{16}). Recuérdese que cada media CAM tiene su propia tabla de búsqueda.

La caja negra de CAM-A se puede visualizar en la Figura 2.3; para el esquema de la caja negra de CAM-B hay que reemplazar el 0 y el 1 en las etiquetas de las salidas por 2 y 3, respectivamente. Las salidas auxiliares (Auxiliar 0 y Auxiliar 1 en la Figura 2.3) se explica más adelante.

En cada media CAM se tiene una caja negra como la de la Figura 2.3,

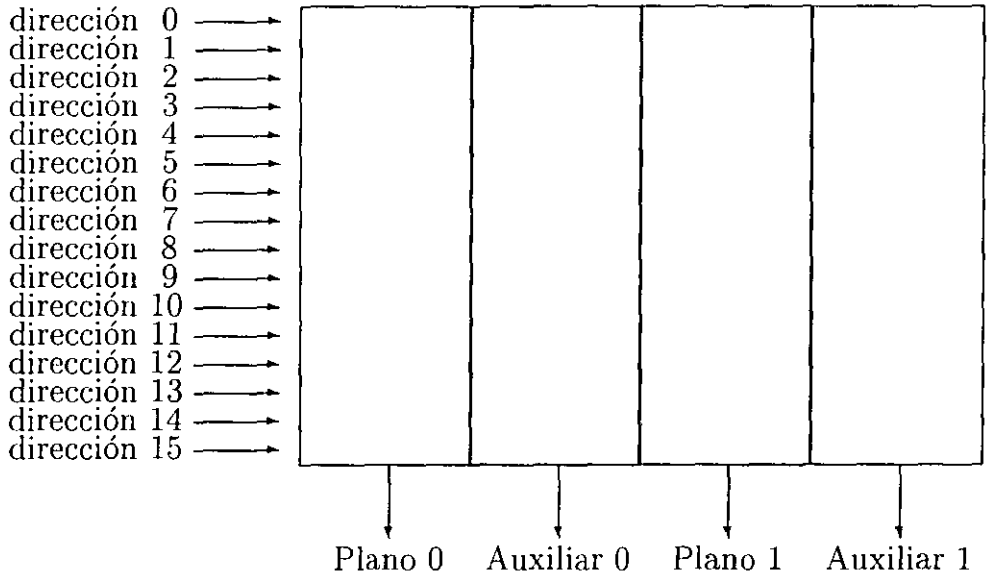


Figura 2.3: Tabla de consulta para CAM-A. Las dieciseis líneas de dirección pueden conectarse a las celdas que se desee conformen una vecindad.

con cuatro puertos de salida y dieciseis cables "pendiendo" de sus conectores de entrada. Podemos imaginar a cada cable como una sonda que puede ser "enganchada" a cualquier variable que queramos percibir. Por ejemplo, para correr Vida necesitaríamos conectar las primeras nueve sondas a las nueve células de la vecindad, y el mapeo (el nuevo estado) se daría únicamente sobre el Plano 0 y sólo necesitaríamos un bit de salida. Si quisiéramos ejecutar un Juego de la Vida con *eco*, donde se muestren con otro color las células que estuvieron vivas una generación anterior, necesitaríamos utilizar otro estado, y por ende, otro bitplano. Así es que tendríamos que utilizar la segunda salida de la tabla para escribir en el Plano 1, ya que la célula para marcar las celdas que han estado vivas anteriormente sería el centro[5] del Plano 1. Hay que notar que en este último caso necesitaríamos utilizar sólo los nueve cables de entrada de un Juego de la Vida normal y no más, puesto que la información necesaria para determinar qué celdas han estado vivas proviene de la célula del centro del Plano 0, que ya tiene una sonda, y no es necesario

[5]El "centro" del Plano 1 es relativo. La célula central es la que en ese momento está siendo evaluada. A la hora de la actualización, cada una de las células irá siendo la central, según se vayan evaluando.

percibir al centro del Plano 1 porque sólo escribiremos en él. El elemento extra que utilizamos (centro del Plano 1) se encuentra en las salidas y no en las entradas. El manejar un autómata celular como Vida o Vida con eco nos deja siete líneas de entrada libres; sin usar. Otra variación está representada por Vida con *rastreo* donde las células que han estado vivas se muestran de forma permanente de otro color. Este último caso es una pequeña variación de Vida con eco.[6]

De la misma manera, los cuatro bits que aparecen en la salida de la tabla pueden ir a diferentes destinos. La primera y tercera columnas de la tabla ("Plano 0" y "Plano 1" de la Figura 2.3), generalmente son usadas para determinar el nuevo estado de los dos bitplanos correspondientes a su mitad de la CAM, y las columnas auxiliares permanecen sin usar. Las auxiliares pueden ser usadas para varias cosas, y en el Capítulo 4 se ven algunos de sus usos.

Debemos hacer la aclaración de que el software de la CAM-PC únicamente reconoce 12 de las 16 líneas de entrada, que es el número de líneas que tenía la CAM-6. No obstante, 12 líneas son suficientes para la mayoría de los experimentos prácticos.

2.2.2 Vecindades dentro de la CAM

En el Capítulo 1 explicamos los elementos de los que constan los autómatas celulares vistos como modelos abstractos. Aquí veremos como son consideradas las vecindades dentro de la computadora CAM-PC.

En la CAM, un vecino "se define" como una fuente de información de un bit, el cual está conectado una de las líneas de dirección de entrada de la tabla de consulta. O sea que un vecino es aquello que está conectado a una de las dieciseis entradas de la caja negra que describimos en la sección anterior. Una vecindad, en este contexto, es entonces una asignación de algunas o todas esas líneas usadas para definir vecinos.

Hemos visto que los procesadores de CAM-A y CAM-B son independientes y que incluso pueden correr distintas reglas cada uno de ellos. De la misma

[6] En general, los dieciseis bits que forman una tabla de consulta pueden ser señales originadas dentro o fuera de la tarjeta CAM. Las que se originan dentro de la tarjeta son extraídas de algunos de los dos bitplanos de una media CAM, de los bitplanos de la otra media CAM, o de la dirección vertical y horizontal de una célula. Las señales de origen externo vienen de la PC, de otras tarjetas CAM o de dispositivos *hardware* como cámaras de video.

manera pueden hacerse distintas asignaciones de vecinos en cada media CAM.

Al programar autómatas celulares en Forth, lenguaje descrito en la siguiente sección, uno no tiene que estar consciente de la asignación de vecindades según la conexión de los cables. Existen instrucciones para definir vecindades que hacen ese trabajo automáticamente.

2.3 El lenguaje de programación Forth

En esta sección daremos una breve introducción al CAM Forth, que de ninguna manera es completa y lo único que se pretende es ilustrar de una manera general la programación de autómatas celulares con Forth. Esta sección es completamente prescindible si sólo se desea utilizar CAMEX para controlar una CAM-PC.

Para aprender a programar en CAM Forth recomendamos la lectura del Apéndice A de [17] o del Capítulo 5 de [2]. Aún así, estas lecturas sólo sirven como una buena introducción a Forth, poque para poder programar en realidad un autómata celular en CAM Forth es necesario conocer cosas que se explican a lo largo de [17] y [2].

El lenguaje estándar para manejar una CAM es Forth, lenguaje desarrollado por Charles Moore. Actualmente Forth está bastante difundido y se puede encontrar en aplicaciones de control en tiempo real y adquisición de datos. Forth incluso fue adoptado como estándar por la International Astronomical Union.

Forth es un lenguaje para tareas interactivas y es adecuado para trabajar con autómatas celulares, y aunque sólo muy poco de Forth es necesario para trabajar con una CAM, ese poco debe ser bien conocido, además de que Forth no es muy parecido a lenguajes de programación convencionales como Pascal o BASIC. Forth es un lenguaje estructurado y —al igual que C— no hace distinción entre procedimientos y funciones.[7]

El intérprete de comandos de Forth separa los datos de entrada en *tokens* (señales, tantos) mediante un espacio en blanco, que en realidad puede ser uno solo o varios seguidos. El programa se ejecuta al presionar la tecla ENTER[8] (o RETURN). Cada conjunto de caracteres acotado por espacios es llamado *palabra*.

[7]Forth es un lenguaje que también es muy usado en el MIT, y quizás esto influyó para incorporarlo.

[8]Los nombres de las teclas se dejan en inglés.

Forth tiene varias palabras almacenadas en un diccionario, y uno puede agregar nuevas palabras al diccionario definiéndolas en base a las ya existentes (la definición de las palabras primitivas están codificadas en el lenguaje de máquina nativo del CPU). La manera de hacer esto es escribir dos puntos (:) al principio de la línea, el siguiente *token* será la nueva palabra que será almacenada y, los *tokens* que le sigan a ésta serán el conjunto de comandos que definirán a la nueva palabra. Cuando definamos una nueva palabra la frase deberá terminar con un punto y coma (;). Cuando una palabra es interpretada y traducida a lenguaje de máquina, está lista para ser ejecutada o ser referencia de subsecuentes definiciones. Por ejemplo, el siguiente fragmento de programa

```
: 3BEEP BEEP BEEP BEEP ;
```

hará que cuando tecleemos 3BEEP y pulsemos la tecla ENTER, la computadora responda emitiendo tres veces un sonido que corresponde a la palabra BEEP.

En Forth, absolutamente cualquier *token* puede ser incluido en el diccionario como una nueva palabra, incluso caracteres que en otros lenguajes son utilizados como signos de puntuación pueden ser parte de una palabra o, un palabra completa por sí mismos.

Además de agregar palabras nuevas al diccionario, una palabra ya existente puede ser redefinida en la misma forma en la que se añaden palabras nuevas, salvo que deberá teclearse la palabra CR entre la palabra a ser redefinida y las nuevas instrucciones que la redefinirán. Ya que cualquier palabra se puede redefinir, no deben de ser problema los nombres crípticos que se usan en Forth, y que tienen su origen en las antiguas computadoras de poca capacidad de donde los nombres cortos eran altamente deseables.

Todos los datos que son introducidos por el usuario, como caracteres, números, variables, etc. son amontonados en una pila (*stack*), y tomados de ahí mismo cuando son requeridos. Esta es la forma en la que funciona Forth, los datos son almacenados en celdas y no tienen etiqueta alguna, el programa siempre toma el dato que se encuentre en la cima de la pila. La forma en que trabaja el programa hace que una dirección permanente no sea necesaria, así es que si un dato que no sea el superior es requerido, el programa hace algo así como levantar las celdas que están encima del dato que se necesite, y entonces tomará el dato, que será entonces el superior, relativamente hablando.

Forth utiliza la notación polaca inversa, conveniente para que las expresiones aritméticas y lógicas de longitud arbitraria sean introducidas sin

necesidad de paréntesis. Cuando se introduce un número, éste se va inmediatamente a la celda superior de la pila. Daremos un ejemplo de como se acumulan los datos en la pila, si nosotros escribimos la expresión:

```
1 2 + 4 + .
```

el número 1 se va a la pila y después el número 2, quedando en la parte superior. La palabra '+' hace que se tomen los valores de las dos celdas superiores de la pila, y que se sustituyan por la suma, que quedará en la parte superior de la pila, por lo que la pila disminuirá en un nivel, ya que los lugares ocupados por el 1 y 2 se sustituyen por un solo lugar cuyo valor será su suma. Después de que el 4 va a la pila y se realiza la segunda suma, la palabra '.' (un punto) hace que se tome la celda superior de la pila y que se despliegue su contenido en la pantalla, quedando la pila nuevamente reducida en un nivel. Ilustraremos la evolución de la pila en la siguiente tabla, observando también cada entrada (*input*) hecha por el usuario así como la salida (*output*) que genera el programa.

PILA	ENTRADA	SALIDA
-	1	
1	2	
1 2	+	
3	4	
3 4	+	
7	.	7
-		

En la tabla anterior se muestra en la columna PILA cómo los elementos se "amontonan" uno sobre otro. El primer elemento de la pila se escribe hasta la izquierda y el que está encima de este a la derecha; así en el tercer renglón el número 2 se encuentra encima del 1 y es el que está hasta arriba de la pila. En la columna ENTRADA se muestra lo que es tecleado por el usuario, y en la columna SALIDA lo que se observa en la pantalla. Como un programa en Forth se ejecuta presionando la tecla ENTER, para ejecutar el programa que hace que suceda lo que se observa en la tabla anterior, debemos teclear 1 2 + 4 . y después presionar ENTER.

Nótese que, por ejemplo, la expresión 1 1 + 1 + 1 + dará el mismo resultado que, digamos, 1 1 1 1 + + +, salvo que la segunda expresión hace que la pila sea temporalmente mayor. Una vez que el resultado —4 en este

caso— es puesto en la pila, no importará mediante qué expresión se llegó a él.

Cuando los compiladores para C o Pascal analizan una expresión como 1 + 2, primero la transforman antes de generar el código de máquina. No es difícil acostumbrarse a escribir las expresiones aritméticas en notación polaca inversa, y esto permite que el compilador sea más simple, más pequeño y más rápido.

Es posible utilizar la pantalla de edición de Forth (o cualquier editor de texto ASCII) para escribir programas que sean guardados, y posteriormente cargados para su ejecución, sin que en este caso el programa sea ejecutado presionando la tecla ENTER. Podrá comprenderse que un programa fuente en Forth no tiene ningún ENTER entre sus líneas.

Es conveniente adoptar un formato para facilitar la comprensión del código y saber qué elementos entran y salen de la pila, por lo que se conviene en escribir las palabras del diccionario en la parte derecha de la pantalla, y las definiciones de éstas en la izquierda. A continuación se muestra una instrucción escrita de la siguiente forma.

```
                                    :  3BEEP
                BEEP BEEP BEEP      ;
```

Como en cualquier lenguaje de programación, es conveniente introducir comentarios en el código que ayuden a la comprensión del programa (como se hace con la instrucción REM de BASIC). En Forth cualquier cadena de caracteres que en una línea de programa aparezcan después de una diagonal invertida (\) será tratada como comentario. Para escribir un comentario en medio de una línea existe la palabra '(', que ignora todo lo que aparezca después, hasta —e incluyendo— el carácter ')'. Después de '(' y antes de ')' debe de aparecer un espacio en blanco para que estos caracteres sean reconocidos como las palabras que se quieren utilizar.

Al igual que en otros lenguajes, podemos definir constantes. Basta con teclear el número deseado seguido de la palabra `CONSTANT` y, después, la palabra que queremos sea equivalente al número. Cuando tecleemos la palabra escogida será lo mismo que teclear el número al cual relacionamos la palabra.

CAM Forth es un lenguaje donde hay variables, iteraciones, y todos los elementos que se encuentran en cualquier lenguaje de programación; pero lo que se pretende en esta sección es solamente ilustrar de manera general la forma en que funciona cuando se trabaja con la CAM-PC. Para mostrar

un ejemplo de aplicación real en la programción de autómatas celulares explicaremos el código principal[9] del Juego de la Vida. El código de Vida en Forth es el siguiente:

```
                                        : 8SUM
                    N.WEST NORTH N.EAST
                      WEST         EAST
                    S.WEST SOUTH S.EAST
                         + + + + + + + ;
                                        : LIFE
8SUM { 0 0 CENTER 1 0 0 0 0 0 } >PLN0 ;
```

Explicaremos, línea a línea el código anterior.

Los dos puntos que aparecen al principio indican que se definirá `8SUM` como una nueva palabra, en base a las palabras que le siguen (`N.WEST`, `NORTH`, etc.) Cada una de estas palabras toma el valor de 0 o 1 según el estado en el que se encuentre cada célula de la vecindad, y esos valores se sumarán todos al escribir siete veces la palabra '+', cuyo funcionamiento ya fue explicado. El punto y coma indica que se ha terminado de definir una nueva palabra. Posteriormente se define una nueva palabra que se llamará `LIFE`. En la definición de `LIFE` tenemos antes que nada a `8SUM`, y el valor de la suma irá a parar encima de la pila. El símbolo de llave que abre es en Forth lo que en lenguaje C es un `switch` y en Pascal un `case`; y la secuencia de palabras que aparecen antes de la llave que cierra son diferentes opciones que tomarán lugar según el criterio de la instrución *case* de Forth. Este criterio se basa en checar el valor que se encuentre en la parte superior de la pila, que en este caso es `8SUM`, y si es 0 toma lugar la opción `0`; si es 1, toma lugar la opción `0`; si es 2, la opción `CENTER`; si es 3, la opción 1, y si el valor de `8SUM` es 4, 5, 6 o 7, toma lugar la opción `0`. Entre las opciones tenemos ceros y unos y la opción `CENTER`, que falta por explicar. La palabra `CENTER` toma el valor de la célula del centro que es la que está siendo evaluada, o sea que su valor puede ser 0 o 1. Así, tenemos que lo que se está haciendo es aplicar la regla de evolución de Vida, donde la celda evaluada morirá si tiene 0 o 1 vecinas vivas; morirá también si tiene 4, 5, 6, 7 u 8 células vecinas vivas; nacerá (o permanecerá viva) si tiene exactamente 3 vecinos vivos; y si tiene 2 vecinos vivos entonces la célula evaluada permanecerá viva si ya estaba viva, y permanecerá muerta si estaba muerta. Para conservar el valor que ya tenía la célula es por lo que

[9]Faltan las declaraciones para definir vecindad, para crear la tabla de consulta, etc.

se usa la palabra CENTER. La opción que tome lugar en la instrucción *case* puede ser 0 o 1 (CENTER también puede ser 0 o 1) y con la siguiente palabra, >PLN0, se asigna al Plano 0 ese valor, que será el nuevo valor que tomará la célula evaluada en la siguiente generación. Finalmente, sigue un punto y coma que indica que ha terminado la definición de la palabra LIFE, por lo que lo que resta es teclear LIFE y oprimir ENTER para ejecutar el programa.

2.4 Otros recursos de la CAM-PC

Se ha expuesto lo que es importante saber sobre la CAM-PC para poder aprovecharla al realizar experimentos con autómatas celulares. Sin embargo la CAM-PC cuenta con algunos recursos adicionales que el usuario pudiera querer utilizar. A continuación se mencionan dichos recursos adicionales.

2.4.1 El mapa de colores

El mapa de colores es una tabla de consulta pequeña que consta de cuatro entradas y cuatro salidas que se hallan relacionadas directamente con los cuatro rayos de color del monitor (intensidad, rojo, verde y azul; IRGB por sus siglas en inglés).

La utilidad principal del mapa de colores es la de adecuar los colores de las células desplegadas en el monitor de acuerdo a los gustos o las necesidades del experimentador. Las entradas de la tabla de consulta del mapa de colores pueden recibir información tanto del Buffer de Planos como de las tablas auxiliares.

2.4.2 El contador de eventos

La CAM-PC incluye un contador de eventos que cumple con la función de proveer capacidades para realizar un análisis del autómata celularen tiempo real. Aunque pudiera ser de utilidad, este contador es muy rudimentario puesto que sólo permite llevar un registro de un sólo evento (un evento puede ser, por ejemplo, el cambio de estado de una célula, y en este caso llevaría la cuenta de todas las células que en un tiempo determinado cambien a un cierto estado).

El contador de eventos está conectado de forma permanente a la salida de intensidad del mapa de colores, y entonces el usuario debe buscar la manera

de caracterizar los eventos que serán contados y asociarlos a esta salida. El momento en que los eventos serán contados depende de la fuente de entrada seleccionada para el mapa de colores.

Para conocer más sobre el contador de eventos, leer la Sección 9.10 de [2] y la 7.9 de [13].

2.4.3 El conector del usuario

Una característica más de la CAM-PC es que cuenta con un conector del usuario (*user connector*) dividido en dos partes, USER 1 y USER 2, que físicamente consisten de dos filas de 25 patas (*pins*) cada uno.

Los conectores del usuario permiten que éste, mediante el uso de conexiones específicas entre los pines, pueda definir sus propias vecindades o incluso pueda conectar componentes externos de *hardware* a la tarjeta.

Todo lo que el usuario necesita saber sobre el manejo de estos conectores puede ser encontrado en el Capítulo 8 de [13] y en el Capítulo 10 de [2].

2.4.4 La vecindad de Margolus

La CAM, aparte de las vecindades de von Neumann y Moore, tiene incorporada la vecindad de Margolus, debida a Norman Margolus, uno de los creadores de la CAM. La vecindad de Margolus es útil para el modelaje de ciertos fenómenos físicos, como el crecimiento dendrítico y difusión de gases.

Esta vecindad está compuesta por bloques de células donde todas las celdas que los componen evolucionan al mismo tiempo, e involucra información referente a las pseudovecindades espaciales y temporales (Sección 4.3.2). Para más información sobre el funcionamiento de esta vecindad, ver el Capítulo 12 de [17].

2.4.5 Autómatas Celulares en 1 y 3 dimensiones

La CAM-PC fue diseñada principalmente para hacer experimentos con autómatas celulares en dos dimensiones (número de dimensiones en los que se centra este trabajo), mas permite emular autómatas celulares en 1 y 3 dimensiones. Para autómatas celulares unidimensionales se pueden realizar estudios mucho más completos en cuanto a su análisis (estadístico, auxiliado con gráficas, etc.) con los programas LCAU, aunque la evolución es mucho más lenta que con la CAM-PC.

Si se desea más información sobre cómo programar autómatas celulares en 1 o 3 dimensiones se pueden consultar [17] y [12].

2.5 Comentarios finales

Ya van 10 años desde que la CAM-PC surgió al mercado, y desde entonces los microprocesadores de las computadoras personales han evolucionado a pasos agigantados. Tendría que hacerse algún estudio serio para saber si las computadoras personales de hoy en día pueden superar a la CAM-PC en la tarea de simular autómatas celulares. Pero independientemente del resultado, sigue teniendo sus ventajas tener un dispositivo completamente dedicado a la simulación de autómatas celulares sin ser parte de la computadora huésped.

Un aspecto por demás interesante respecto a la CAM-PC, es que su diseño está inspirado en los mismos autómatas celulares, es decir, es un ejemplo de cómo modelos teóricos de computación pueden acabar en nuevos paradigmas de construcción y diseño de dispositivos electrónicos de cálculo. No sería raro que los autómatas celulares tuvieran algo que ver en nuevas computadoras paralelas, si es que alguna vez surge la tan ansiada quinta generación.

Capítulo 3

El entorno del programa CAMEX

Teniendo un buen conocimiento del *hardware* de la CAM-PC, es posible para los programadores crear sus propios programas que controlen la tarjeta e interactúen con el usuario. Ese es precisamente el caso de CAMEX (*CAM-PC Exerciser*, desarrollado por Harold V. McIntosh en la Universidad Autónoma de Puebla (UAP)), programa al cual nos introduciremos en el presente capítulo. El rango de estados y la longitud de vecindades a las cuales CAMEX está adaptado es una función del *hardware* de la CAM, que fue construida para manejar sólo algunas posibilidades —de entre todas las existentes— de vecindades y número de estados.

CAMEX está escrito en lenguaje C, lenguaje muy difundido (mucho más que Forth) donde se puede programar a bajo nivel, y por ende controlar un dispositivo *hardware* como una tarjeta CAM-PC. Además la portabilidad de este lenguaje lo hace conveniente en el caso de cualquier cambio que se pueda llegar a producir en la computadora huésped. Con el programa se distribuye también el código fuente, haciendo posible para cualquier programador el modificar el programa tanto como desee, ya sea programando más reglas y configuraciones de autómatas celulares, interfaces más amigables, y en general, cualquier modificación o adición necesaria para un mejor manejo de la tarjeta o para algún estudio particular.

CAMEX es en cierto modo la continuación de la familia de programas LCAU (Linear Cellular Automata)[1], colección de programas que, debido a

[1]Estos programas evolucionaron en la UAP, y aparte la evolución del autómata, muestran el cálculo de los diagramas de de Bruijn, un análisis de probabilidades de equilibrio,

la cantidad de computación requerida en los autómatas celulares, se centraron en autómatas celulares de una dimensión. Con la disponibilidad de un controlador de video como la CAM-PC, pueden vislumbrarse estudios más ambiciosos en autómatas celulares bidimensionales, que son lo "natural" en la CAM, puesto que para eso fue diseñada [17].

CAMEX es un programa que, como ya se mencionó, aparte de ser de distribución gratuita es de código abierto (*open-source*), lo que significa que los archivos fuente con los que CAMEX fue creado se distribuyen libremente, y eso brinda grandes posibilidades a los usuarios de CAMEX, de poder reprogramarlo en la forma que les convenga. Aún más, es posible que en un futuro CAMEX sea desarrollado de forma común entre varios programadores.

De lo anterior debe entenderse que CAMEX sólo está limitado, prácticamente, por el *hardware*. Pero el poder que este programa brinda a los usuarios programadores involucra un conocimiento profundo de la manera en que funciona la CAM-PC, así como la estructura del programa CAMEX. En el Capítulo 2 ya se habló sobre el *hardware* de la CAM, y en el Capítulo 4 se proporciona más información sobre el *hardware* y sobre el programa CAMEX.

En este capítulo se ilustra al lector sobre las tareas y actividades que se pueden llevar a cabo desde la interfaz que CAMEX proporciona al usuario. Como en cualquier otro caso similar, la información aquí presentada sólo podrá ser aprovechada en su totalidad si se tiene disponible una CAM-PC.

3.1 La interfaz de CAMEX

Se ilustrará cómo el usuario puede controlar la tarjeta CAM-PC a través del programa CAMEX.

La interfaz que CAMEX nos proporciona no es la más amigable si la comparamos con los paquetes comerciales para microcomputadoras porque CAMEX fue escrito antes de la época de la programación visual con ventanas, pero después de saber cómo manejarla, se antoja realmente intuitiva; diseñada de una manera en que el manejo de ésta es sumamente práctica y que hace que el diseño de experimentos con autómatas celulares sea una actividad que el usuario siente que realiza de una manera muy eficiente. Después de todo, CAMEX está pensado como una opción para controlar tarjetas CAM-PC y está dirigido a los programadores y estudiosos que quieran aprovecharlo; no

y calculan ancestros. Se pueden encontrar (junto con versiones para OpenSTEP y NeXT, llamadas NXLCAU) en la página http://delta.cs.cinvestav.mx/~mcintosh.

pretende causar una buena impresión siguiendo los lineamientos del *software* comercial.

Después de cargar el programa en la memoria tecleando el nombre del archivo ejecutable (`CAMEX.EXE`, por ejemplo), lo primero que observamos es la nota del *copyright* inscrita en un rectángulo en la parte derecha de la pantalla, y sólo es visible al inicio del programa. Esa misma área es en donde se despliegan los dos páneles de ayuda con que cuenta el programa.

Cada uno de los dos paneles de ayuda aparecen con las teclas '?' y '!', respectivamente. El panel asociado a '?' contiene una lista de las distintas opciones disponibles presionando letras del teclado estándar, y el panel invocado por '!' explica el uso de las teclas de función.

Se ha tratado, hasta donde ha sido posible, que todas las opciones de los menús de CAMEX están disponibles durante todo el tiempo. Esto hace muy cómodo el uso del paquete, al no variar casi en nada el significado de las teclas según el contexto, al estar dentro de algún submenú.

Una característica curiosa de CAMEX, es que cuando inicia no hace ninguna cosa tal como limpiar bitplanos, o quitar la regla del Procesador de la CAM. Así es que toda la información de alguna sesión anterior en la que se haya utilizado la tarjeta CAM-PC —bitplanos, tabla de consulta y mapa de colores— permanece en ella y se puede seguir trabajando con esa misma información[2]. Entre las sesiones anteriores, se incluyen las sesiones en las que se haya trabajado con el *software* original de la CAM. Lo que es más, la información en la tarjeta permanece aún reiniciando la máquina, y solo se pierde cuando la computadora huésped es apagada y la corriente eléctrica deja de circular.

Se puede aprovechar la característica anterior y tomarla como ventaja para, por ejemplo, insertar una nueva regla sobre una configuración arrojada por una regla anterior. Pero si por cualquier motivo no se desea que la tarjeta guarde la información, es fácil borrar los bitplanos y regresar al mapa de colores original de CAMEX desde el entorno integrado. (Si se deseara, podría reprogramarse CAMEX para que inicie con ciertos valores predeterminados).

[2]No ocurre lo mismo con el *software* original, donde al iniciar una nueva sesión los valores dentro de la CAM son puestos de una forma predeterminada.

3.2 El menú principal

El menú principal de CAMEX, desplegado con '?', contiene los siguentes elementos:

```
- options - keyboard menu -
function keys - see ! panel

S,s -  Run on, single step
q - return to DOS
z#,u# - clear,set bitplanes
p#,R#,c#,w# - random points
H#,h# - make checkerboard
(# = hex bitplane select'n)
y - random ellipse plane 0
vx - permute the planes
     (x=0,1,2,3,C,c,X,x,l)
d - de Bruijn diagram
r - edit rule (see f1)
l - edit bitplane (see f1)

n# (#=nsew) shift plane 0
N# (#=nsew) shift all pl.

f1 selects rule type; rule
can be edited with r, plane
with l (if appropriate).

others --- see source code
```

A continuación se explica el funcionamiento de las opciones de la lista anterior.

3.2.1 Corriendo la CAM

Para que la CAM dé un paso en la evolución de algún autómata celular basta con presionar la tecla s. La evolución se realizará de acuerdo a la regla que en ese momento se encuentre en el Procesador de la CAM, y con la configuración actual en el Buffer de Planos (que generalmente se visualiza en la pantalla). Al teclear la letra mayúscula S el autómata celular seguirá corriendo repetidamente hasta que alguna tecla sea presionada.

3.2.2 Salir del programa

Se abandona el programa tecleando `q`, mientras que para salir de cualquier subrutina del programa, basta con presionar la tecla ENTER.

3.2.3 Bitplanos

Las letras `z` y `u` operan sobre los planos de bits limpiándolos (poniendo en ceros todos los bits) y llenándolos (escribiendo unos en todos los bits) respectivamente. Estas dos letras necesitan parámetros de opciones. Dichos parámetros son números hexadecimales (de 0 a f) y sobre ellos se hablará en detalle en la Sección 4.2.1.1, ya que están estrechamente relacionados con el *hardware* de la tarjeta. Al teclear alguna de estas letras el programa espera recibir otro carácter, y la acción toma lugar hasta que las dos teclas son presionadas (la primera es la tecla de la letra con una acción asociada, y la segunda tecla corresponde al argumento). Según el argumento que se proporcione, se puede elegir el o los planos en los que se realizará la escritura. Puede elegirse uno de los cuatro planos o cualquier combinación de éstos. Como se acaba de mencionar, se indica en que plano o planos se escribirá mediante un número hexadecimal y si se quiere tener una referencia completa de la forma en que operan estos números deberá verse la tabla 4.10. Por ejemplo, si se quieren limpiar todos los planos deberá teclearse `z` y continuación la letra `f` (`zf`) y si se quiere escribir (poner en 1s) en todos los planos entonces deberá teclearse `uf`. Basándonos en la tabla referida, veremos que para seleccionar los planos 0, 1 y 2 se tecleará como argumento `7` y el argumento `c` para operar sobre los planos 2 y 3, por mencionar algunas combinaciones. Es posible utilizar el argumento `0` aunque en este caso no se seleccionaría ningún plano.

El diseño del *hardware* de la CAM permite escribir en múltiples planos a la vez, pero limita la lectura a un solo plano. Como CAMEX se programó siguiendo rigurosamente al *hardware* de la CAM-PC, se tiene un parámetro hexadecimal entre `0` y `f` para la escritura en planos, pero se utiliza un parámetro decimal entre `0` y `3` para la lectura. La lectura y escritura de planos, así como el significado de cada uno de los argumentos hexadecimales, se explica en la Sección 4.2.1.

Las letras `p`, `R`, `c` y `w`, que también tienen parámetros, introducen puntos aleatorios en los planos de bits. En el orden en que se mencionaron, lo que hacen es introducir puntos con densidad baja, densidad media, densidad alta, y puntos escasos.

La proporción de puntos de cada una de las tres densidades, se puede personalizar en el menú de parámetros que se verá en la Sección 3.4.3. Para especificar la proporción deseada se selecciona un número entre 0 y 999; y ya que se considera una totalidad de mil, el número 500, por decir algo, significará un 50% de probabilidad ($500 \div 1000$). El parámetro de los puntos escasos (también entre 0 y 999) se refiere a un número fijo de puntos, distribuidos aleatoriamente en la pantalla.

Las opciones `H` y `h`, que también requieren de parámetros, ponen un "tablero de ajedrez" en la pantalla (la primera coloca un tablero de 2×2, y la segunda toma el valor establecido en el menú de parámetros). Si por ejemplo, quisiéramos dividir la pantalla en cuatro cuadrados en el Plano 1 escribiríamos como argumento `1` habiendo seleccionado previamente el número 4 en el menú de parámetros del que se habla en la Sección 3.4.3.

La opción `y` instala una elipse aleatoria en el plano 0. En una elipse aleatoria la densidad de los puntos que la forman depende de una distribución de probabilidad. La densidad y radio de la elipse son parámetros del menú de parámetros.

Todas estas manipulaciones de los planos proveen de una gran variedad de configuraciones iniciales, o de configuraciones para introducir *ruido* durante la evolución de una autómata celular.

3.2.4 De Bruijn

Este submenú, invocado por `d`, concierne a los diagramas de De Bruijn, una herramienta gráfica con la que se pueden estudiar autómatas celulares no tratada en este trabajo.

3.2.5 Edición de reglas y bitplanos

Para algunos autómatas celulares de la lista desplegada con `f1` (Sección 3.4), existen submenús que permiten editar[3] la regla de evolución y la configuración de los planos de bits, con las teclas `r` y `l`, respectivamente. Puede verse qué reglas de autómatas celulares permiten ser editadas en el archivo fuente `CAM.C` después del comentario que dice "`Edit the bitplanes, in keeping`

[3]La edición de la regla de evolución permite especificar de manera detallada la asignación de valores a las vecindades de nuestro autómata. La edición de planos permite asignar los valores deseados en cada célula del espacio, permitiendo la realización de configuraciones iniciales muy específicas.

`with the type of rule which has been selected.`" Las reglas que permiten la edición son aquellas en las que la línea en que se encuentran, se llama a una función antes de la sentencia `break`, como por ejemplo en la siguiente línea:

```
case 14: edmline(arr1,arr2); break;   /* random (2,1) Moore */
```

Las reglas que no permiten edición, que son la mayoría, son como la siguiente:

```
case  0: break;   /* table for Life evolution */
```

Debemos advertir que esta opción no funciona correctamente.

3.2.6 Desplazamientos (*shifting*)

Primeramente hay que aclarar qué es un desplazamiento en CAMEX (o en la CAM, en general). Recordemos que los bitplanos están diseñados como si estuvieran sobre la figura de un toro. Entonces, un desplazamiento sería recorrer los valores de esos bitplanos sobre la superficie toroidal. Por ejemplo, si desplazamos el Plano 0 un espacio hacia el norte, entonces cada renglón se movería hacia el renglón arriba de él, copiándose los 1s y 0s hacia el renglón superior, y debido al toro, el renglón de hasta arriba pasaría a ser el inferior.

Se pueden realizar desplazamientos de los planos con `n` (plano 0) y `N` (todos los planos), y los argumentos de estas dos direcciones son `n`, `s`, `e` y `w`, que especifican las direcciones de los desplazamientos (norte, sur, este, y oeste). Después de haber tecleado las dos letras para un desplazamiento (la letra que indica que se realizará un desplazamiento y su argumento direccional) el desplazamiento se efectúa en realidad a la hora de presionar `s` o `S`, igual que si se estuviera ejecutando una regla. Si por ejemplo quisiéramos desplazar todos los planos cinco posiciones hacia el norte deberemos teclear `N` seguido de la letra `n` y luego `s` cinco veces.

El inconveniente que tienen los desplazamientos en CAMEX, es que se pierde la regla que estaba antes del momento de indicar el desplazamiento, y es necesario volverla a cargar; usualmente presionar INSERT resuelve el problema.

3.2.7 Permutaciones

Con la opción v es posible permutar y complementar planos, de acuerdo a los argumentos que se muestran en la siguiente lista:

0 - complementa el plano 0
1 - intercambia los planos 0 y 1
2 - intercambia los planos 0 y 2
3 - intercambia los planos 0 y 3
c - permutación cíclica
C - permutación anticíclica
x - intercambia pares e impares (El Plano 0 por el 1 y el 2 por el 3)
X - intercambia CAM-A y CAM-B
l - ciclo 'más largo': "Dabc"[4]

Nótese que al complementar planos cambian los estados de las células, debido a que cambiarían las combinaciones de 1s y 0s que, cual número binario, definen el estado en que una célula se encuentra. Recuérdese que en la CAM, los estados de las células están dados por las distintas combinaciones de los valores binarios en cada uno de los bitplanos, por lo que si, por ejemplo, tenemos una celda cuyo estado está dado por 0011 (cero en los planos 0 y 1, y uno en los planos 2 y 3) y complementamos el Plano 0, entonces tendremos 1011 que representará otro estado distinto. Puede observarse que el cambio de estado al cambiar las combinaciones de unos y ceros también se dará al permutar planos.

La opción c, permutación cíclica, simplemente intercambia cada plano con el que le sucede, es decir, los datos en el Plano 0 son copiados al Plano 1, mientras que los datos de éste pasan al Plano 2, cuyos datos son copiados en el Plano 3. Los datos de este último plano pasarán al Plano 0. La permutación anticíclica (opción C) copia los datos de los planos en un orden inverso al explicado previamente.

[4]Este ciclo es parecido a la permutación cíclica, con la salvedad de que los datos del Plano 3 son complementados mientras son copiados en el Plano 0. Esto también se halla explicado en la Sección 5.2 de [12].

3.3 Otras opciones del entorno

Existen algunas opciones que no se despliegan con '?', pero que se pueden ver en el archivo fuente `CAM.C`, dentro de la función `camex` (que a su vez es el cuerpo de la función `main`, que también se encuentra en el mismo archivo). La lista completa de opciones se encuentra después del comentario que dice "`Alternatives corresponding to function keys`". La razón de no describir todas las opciones, es que no están bien documentadas y empíricamente es difícil ver su función con exactitud. Probablemente esto sea código muerto, que quedó como muestra de las modificaciones que pensaban hacerse a CAMEX en el futuro. Tómese en cuenta también que al ser CAMEX un programa con el cual se distribuye libremente el código fuente, la labor de verificar y corregir la función de las opciones no descritas puede ser una tarea para futuros desarrollos sobre CAMEX. A continuación se describen algunas de estas opciones:

- La tecla `~` complementa planos, es decir, cambia los bits con valor 1 por 0 y viceversa. Para indicar qué planos complementará se da un argumento que se teclea después de presionar `~` . (Recuérdese que para escribir en planos se usa un argumento hexadecimal entre `0` y `f`).

- La tecla `f` hace que los planos 0, 1, 2 y 3 se desplacen hacia el norte, este, sur y oeste, respectivamente. El desplazamiento toma lugar al correr la CAM con `s` o `S`. Al desplazar los plano, lo que sucede es que los estados de todas las células que conforman ese plano, se recorren simultáneamente en alguna dirección. Esta función es particularmente útil para centrar o mover de lugar configuraciones, sobre cuando se inicia CAMEX ya que siempre mueve la configuración que se encuentra en el buffer de planos.

- La tecla `F` desplaza lo que hay en el Plano 0 nueve espacios (pixeles) al sur, nueve al este, nueve al sur, nueve más al sur, y siete al oeste; dejando rastro en el Plano 1.

- La tecla `L` es una opción muy práctica. Instala la regla de Vida en el Plano 0, con eco en el Plano 1.

- La tecla `o` hace un "OR" de los planos 0 y 1, colocándose el resultado de esta operación en el Plano 0.

- La tecla P pone un patrón (configuración ya predeterminada en CAMEX) en los planos que se le indiquen con un argumento.
- La tecla V instala una línea de ceros en un campo de unos (alguna región donde todas las celdas tienen el valor de 1). La línea es la columna de hasta la izquierda y mediante un argumento se indican los planos sobre los que se actuará.
- La tecla Y pone puntos aleatorios (con 50% de probabilidad) en el Plano 0.

3.4 Las teclas de función

El uso principal de las teclas de función es el desplegar menús adicionales. Aparte del menú principal desplegado por '?', hay un menú de teclas de función que responde a '!'. Este menú aparece también en el panel derecho y su contenido es el siguiente:

```
  - use of function keys -
f1  - rule or bitplane menu
f2  - rec demonstrations
f3  - edit rec expression
f4  - execute f3 expression
f5  - execute f2 expression

f7  - parameters & values
f8  - alternate INSERT
f9  - show bitplanes
f10 - change color palette

^X - go to extreme X value
@X - reverse X's direction
insert - install f1 option
erase  -  clear all planes
up/down - f1 sequence (ins)
page u/d - f2 sequence (f5)
arrow l/r - parameter seq
arrow u/d - incr/decr par

? - keyboard options
```

3.4.1 El menú de autómatas

Con `f1` se muestra una colección de 38 elementos que consiste en reglas y configuraciones iniciales. Se puede "navegar" por esta lista con las teclas de dirección, con '<' se posiciona el cursor en la parte superior de la pantalla y con '>' en la inferior. Para seleccionar una opción de este menú se posiciona el cursor en alguna opción y se presiona la tecla INSERT. Puede presionarse ENTER antes de INSERT para visualizar únicamente la regla seleccionada. Cuando se elige una regla presionando INSERT, no se ejecuta en ese momento (sólo se inserta en la CAM); y para correr la regla se deberá presionar `s` o `S`. Haremos notar que si se inserta una regla mediante INSERT sin haber presionado ENTER previamente tendremos que salir del submenú de `f1` antes de correr la regla.

3.4.2 El menú de REC

Esta segunda colección consiste en demostraciones escritas en el lenguaje de programación REC. Hablaremos de este lenguaje y del papel de las teclas `f2`, `f3`, `f4` y `f5` en la Sección 3.6.

3.4.3 El menú de parámetros

Es en este menú, asociado a `f7`, donde se pueden modificar los parámetros de muchas de las opciones con las que cuenta el entorno de CAMEX. Consiste de los siguientes elementos:

```
 1: atern (    ) alternate rule
 2: wfrno (    ) Wolfram rule number
 3: hdens (    ) high density - mils
 4: mdens (    ) medium density - mils
 5: ldens (    ) low density - mils
 6: wdens (    ) scarce dots - number
 7: elrad (    ) radius random ellipse
 8: nchek (    ) ch. squares per row
 9: tacol (    ) table color
10: macol (    ) marker color
11: cucol (    ) cursor color
```

El movimiento a través de estas opciones se realiza (verticalmente) con las teclas de dirección, mientras que con las teclas horizontales (izquierda y

derecha) se disminuye e incrementa el valor del parámetro, respectivamente. El valor del parámetro aparece dentro de los paréntesis. ('<' y '>' funcionan igual que en `f1`). A continuación sigue una descripción de cada opción en este menú.

1. atern - Escoge tablas de consulta alternas para las vecindades de Moore cuando existen; su rango es de 0 a 3. Las tablas de consulta alternas están contenidas en las tablas auxiliares (ver Sección 2.2.1), y aquí hay que señalar que para conocer las reglas almacenadas en las tablas alternas es casi indispensable conocer el código fuente por lo que no es posible dar mayores explicaciones sobre este aspecto en este capítulo y se verá en la Sección 4.3.2.1).

2. wfrno - El parámetro de 0 a 255 selecciona la regla decimal de Wolfram (Sección 1.4.1) para la opción de autómatas celulares unidimensionales (2,1).

3. hdens - Establece la densidad de células en estado 1 para la opción `c`. (Ver Sección 3.2.3).

4. mdens - Establece la densidad de células en estado 1 para la opción `R`. (Ver Sección 3.2.3).

5. ldens - Establece la densidad de células en estado 1 para la opción `p`. (Ver Sección 3.2.3).

6. wdens - Especifica el número de puntos para la opción `w`. (Ver Sección 3.2.3).

7. elrad - Con un parámetro de 0 a 54, especifica el radio (en unidades de unos 6 pixeles) de la elipse de la opción `y`. (Ver Sección 3.2.3).

8. ncheck - Indica el número de cuadrados del tablero de ajedrez de la opción `h`.(Ver Sección 3.2.3).

9. tacol- El número entre 0 y 255 usado por `videocattr` para desplegar entradas de la tabla.

10. macol - El número entre 0 y 255 usado por `videocattr` para marcar entradas de la tabla.

11. cucol - El número entre 0 y 255 usado por `videocattr` como cursor cuando despliega tablas.

Las últimas tres opciones están relacionadas con la opción de editar reglas (Sección 3.2.5).

3.5 Otras teclas de función

Existen más tareas asociadas a otras teclas de función que no aparecen listadas al desplegar el menú '!', y a continuación se comentan algunas de ellas (no se explican en su totalidad por las mismas razones que se dieron en la sección anterior). Las funciones de `f9` y `f10` (la función `f8` no funciona) también se describen a continuación, aunque éstas sí aparecen en el menú '!'. La lista completa se encuentra en el archivo fuente `CAM.C` después del comentario que dice "`Alternatives corresponding to function keys`".

- Presionando repetidamente la tecla `f9` se muestran en el monitor, uno a uno, cada uno de los 4 planos de bits en orden consecutivo (planos 0, 1, 2, 3, y nuevamente el 0). Por ejemplo, si estuviéramos corriendo el autómata celular del Juego de la Vida con rastro, al presionar una vez `f9` veremos solamente la regla del autómatas celulares, y al apretar la tecla por segunda vez veremos únicamente la "mancha" dejada por el rastro.

- Con la tecla `f10` se muestran consecutivamente las diferentes combinaciones de la paleta de colores de CAMEX. De ninguna manera son todas las combinaciones que pueden generarse en la CAM-PC, sino un conjunto arbitrario de combinaciones.

- Al presionar `^f9` (dejando oprimida la tecla CONTROL mientras se oprime `f9`) se muestra, en el monitor, únicamente el Plano 0.

- Con `@f9` (presionando primero `@` y después, como si fuera un argumento, `f9`) se muestran uno por uno los planos de bits, pero en orden inverso (3, 2, 1,0 y otra vez 3) al normal, mostrado con `f9`.

- Una tecla muy útil es `F9` (presionando `f9` mientras se mantiene apretada la tecla SHIFT), que establece los colores primarios de los planos. Esta opción es muy recomendable cuando se experimenta con escritura en

planos o con instrucciones que manipulen planos, pues por los colores asignados de esta forma, es fácil reconocer las configuraciones existentes en cada bitplano.

3.6 El lenguaje de programación REC

El lenguaje que se utiliza en CAMEX para programar autómatas celulares es REC (*Regular Expression Compiler*), también desarrollado por Harold V. McIntosh. El lenguaje REC, como tal, surgió en 1966 en la Universidad Nacional Autónoma de México, al implementarse en una computadora PDP-8 (aunque este lenguaje se gestó en la Universidad de Florida, ligado a LISP); posteriormente se usó REC por más de una década en el Instituto Politécnico Nacional, en numerosas aplicaciones; y en el Instituto Mexicano de Energía Nuclear.

La programación de autómatas celulares con el REC de CAMEX es una tarea muy sencilla, ya que hay aspectos generales de REC que no se utilizan dentro de CAMEX. Si se desea tener una comprensión mucho más completa del lenguaje REC en general puede consultarse [22].

3.6.1 El lenguaje

REC, más que un lenguaje de programación es una estructura de control. La esencia de REC radica en el uso de cuatro símbolos de control, que son los paréntesis, los dos puntos y el punto y coma (`(`, `)`, `:` y `;`), junto con operadores y predicados. Los operadores son simplemente subrutinas, y los predicados tienen, además, un valor de verdad. Los operadores y predicados pueden ser de tres tipos: sin argumento, con un argumento o con dos argumentos. Como es lo usual, un par balanceado de paréntesis sirve para agrupar símbolos y todos los programas en REC deben estar dentro de paréntesis. El flujo del programa se realiza de izquierda a derecha, y se puede ver alterado por símbolos, predicados, por dos puntos, o por un punto y coma. Como en la mayoría de los lenguajes de programación, en REC se ignoran los espacios y tabulaciones mientras éstos no estén entre comillas; la tecla ENTER, como en todo el entorno de CAMEX, nos devuelve a una capa superior del programa (es decir, nos saca del par de paréntesis en que nos encontramos). También existen los comentarios —útiles para deshabilitar un segmento de código sin removerlo o para documentar el programa—, y todo lo que se encuentre entre

paréntesis cuadrados o corchetes será ignorado.

El carácter ':' (dos puntos) es la iteración en REC, e implica la repetición del código que está entre el paréntesis izquierdo más cercano a la izquierda de ':' y los mismos dos puntos. En el ejemplo que se muestra unos párrafos más adelante se ilustra concretamente el uso de los dos puntos. Por otro lado, el símbolo ';' hace que el programa "salte" hasta el carácter que se encuentre inmediatamente después del primer paréntesis derecho que se halle a la derecha del punto y coma (es decir, termina la expresión actual), asignando el valor de "verdadero" al proceso que se acaba de completar (el que se encuentra entre los paréntesis). Cuando durante el flujo del programa nos encontramos el símbolo), también se termina la subexpresión, pero se asigna a ésta el valor de "falso".

Cuando el programa se encuentra con un predicado se evalúa su valor; si es verdadero, el flujo del programa continúa con la secuencia de símbolos sin interrupción, pero si es falso entonces se saltará hasta el siguiente segmento del programa. El nuevo segmento será aquel que siga al ':', ';' o ')' más cercano hacia la derecha de donde se encuentre dicho predicado. En realidad, los valores de verdad de los predicados no tienen ninguna importancia al programar autómatas celulares en CAMEX, en donde casi todo lo que se maneja son operadores. Existe un predicado útil, denotado por `!n!`, que actúa como contador, y que es verdadero n veces y luego falso. Por ejemplo, si escribimos `(!13!x:;)`, se realizará 13 veces la operación `x`.

En el REC de CAMEX los operadores y predicados están representados por un carácter ASCII. Se pueden ver los caracteres ASCII que están "dados de alta" (tienen significado) y su función en una tabla de símbolos que se encuentra en el archivo fuente `CAMTBL.H`. Los argumentos que puedan llegar a tener los operadores y predicados constan, también, de una sóla letra y deben de ser tecleados inmediatamente después de la letra asociada al operador o predicado. La tabla de símbolos referida es demasiado larga (contiene casi todos los caracteres ASCII) como para que sea visualmente práctico mostrarla completa en este documento, siendo más práctico consultar el archivo que la contiene. De cualquier forma incluimos aquí una parte a manera de ilustración:

```
r_code,     ropla,     "a - pantograph                    ",
r_oper1,    rosdt,     "b - Squares, diamonds & triangles ",
r_oper1,    roplc,     "c - Lichens (0-simple, 1-w/death) ",
r_oper1,    ropld,     "d - dot in center of plane        ",
```

```
r_code,      rophb,      "e - Hollow/Border                ",
r_code,      ropcy,      "f - Cycle for Hollow/Border      ",
```

Cada operador de REC es una subrutina ya compilada en C. Esto nos brinda una gran flexibilidad, porque podemos asignar a una letra ASCII cualquier tarea o conjunto de tareas que deseemos se lleven a cabo, y posteriormente interpretarlas en tiempo real en conjunto con otras instrucciones de REC. La manera de insertar un nuevo operador de REC se discute en la Sección 4.5.

Mientras tanto se muestra y explica un ejemplo de un programa en REC:

```
(zf w1 G (gk:;) ;)
```

donde

- `z` es un operador que con el argumento `f` limpia todos los planos de la CAM;
- `w` es un operador que al tomar como argumento el valor de 1 coloca algunos puntos dispersos en el Plano 0;
- `G` es el operador que instala la regla para el autómata celularde Greenberg-Hastings; y
- `(gk:;)` es un ciclo *while* en el que se repite el operador `g` mientras que no se cumpla con la condición del predicado `k`. El operador `g` da un paso de la evolución en tanto que `k` monitorea la actividad del teclado y termina con el ciclo cuando alguna tecla es presionada. Al presionar alguna tecla, `k` (que tiene el valor de verdadero al iniciar el programa) toma el valor de falso y provoca que el flujo del programa salte al siguiente segmento del código, que en este caso es el fin del programa.

En el ejemplo anterior la configuración inicial fue establecida de manera aleatoria, aunque existe la posibilidad de utilizar en nuestros programas configuraciones previamente establecidas leyéndolas desde un archivo. Dichos archivos son del tipo **.PAT**[5].

[5]Actualmente, CAMEX no cuenta con una opción que permita crear archivos que contengan patrones (archivos .PAT), por lo que de momento éstos deberán ser creados con el *software* original y deben encontrarse en el mismo directorio que el ejecutable de CAMEX.

Una variación de nuestro ejemplo en la que en lugar de una configuración aleatoria usamos un cuadrado formado por células en estado 1 en el Plano 0 sería:

```
(zf "CUADRO.PAT " y1 G (gk:;) ;)
```

donde el operador y al tomar el valor de 1 como argumento coloca en el Plano 0 la configuración contenida en el archivo CUADRO.PAT. El nombre del archivo debe estar entre comillas, y un espacio en blanco debe existir entre el nombre del archivo y las comillas que cierran.

3.6.2 Manejando REC desde el entorno

Con la tecla f2 se despliega una colección de demostraciones —similares a las de f1— escritas en REC. Dichas reglas son ejecutadas presionando la tecla f5, y los programas en REC pueden ser editados mediante f3, pudiendo modificar la demostración elegida o escribir un programa nuevo. Las expresiones editadas por medio de f3 se ejecutan con la tecla f4.

El menú de REC consta de los siguientes elementos:

```
1 - null program - to edit
2 - free flowing evolution
3 - clear plane 1 regularly
4 - reversible & reverse
5 - life on a checkerboard
6 - life with glider trace
7 - life with simple trace
8 - diag reflect pl 0, 1
9 - adiag reflect pl 0, 1
10 - horiz reflect pl 0, 1
```

Para crear un archivo .PAT con el *software* original, al tener la imagen deseada en el monitor, se debe seguir el procedimiento explicado en la guía del usuario de la CAM-PC [2], por medio del cual se guarda la información del Buffer de Planos en un archivo.

Existe un formato muy difundido de archivos con extensión .LIF y cada uno de los cuales contiene configuraciones iniciales del Juego de la Vida. Existe en CAMEX una opción a ser desarrollada, con la cual se puedan leer este tipo de patrones. Mientras tanto, dado que muchos de tales patrones contienen configuraciones iniciales muy interesantes y que pueden ser leídas por los programas *WinLife* y *Life32*, damos la dirección de Internet de donde puede obtenerse la colección de archivos .LIF, recopilados por Alan Hensel.
http://www.mindspring.com/~alanh/life/

```
11 - vert reflect pl 0, 1
12 - rotate by shearing
13 - rotate backwards
14 - load std bitplane pl.0
15 - interesting eater cycle
16 - Silverman rule
17 - Life glider demo
18 - spiral for bubbles
19 - gridwork from points
20 - two-color Life
```

Como su nombre lo dice, las opciones anteriores son simplemente demostraciones de programas de REC. Si lo desea, el programador puede experimentar con ellas y basarse en alguna de ellas si le pudiera servir para algún fin particular. Algunas opciones se explican más detalladamente en [12].

1. `( ;;)` La opción más práctica a primera vista podría ser ésta, ya que sólo contiene las instrucciones que debe llevar todo (o casi todo) programa en REC, que son las instrucciones de iteración y paréntesis balanceados, para que el usuario inserte los operadores de configuraciones iniciales, reglas y vecindades que desee.

2. `(zfR(gk:;);)` Esta opción únicamente pone una elipse como configuración inicial y corre con la regla que en ese momento se encuentre cargada en las tablas de consulta.

3. `(zfR((!200!gk: z2;)k:;);)` Con esta opción se podrá observar una mancha que crece continuamente, y que a cada 200 pasos de evolución el Plano 1 es limpiado.

4. `(zfR(L(!100!gk:;)M(!100!gk:;)k:;);)` Aquí, la elipse inicial comienza a crecer durante 100 con la regla instalada por el operador `L`, para después regresar a su estado original corriendo durante otros 100 pasos con la regla inversa `L` que es instalada por el operador `M`.

5. `(zf n1 h2 14 (g:;);)` Después de limpiar los planos, se comienza desde una configuración inicial de puntos esparcidos (`n`) para luego poner una cuadrícula con `h`. El operador `1` instala la regla de Vida y así obtenemos un Juego de la Vida corriendo en un tablero de ajedrez.

6. `(zf R l0 (gk:;);)` El Juego de la Vida con rastro de deslizadores.

7. `(zf R m0 (gk:;);)` El Juego de la Vida con rastro simple.

8. `(v2 g d1 id (!128!g:;) v0 g v2 g iW (!256!g:;) v1 g;)` Esta opción dibuja una línea diagonal desde el extremo inferior izquierdo al extremo superior derecho de la pantalla y después hace que la configuración inicial se desplace hacia la diagonal, en donde cada partícula (célula) de la configuración, al hacer contacto con la diagonal, cambiará de dirección, para que tengamos al final la misma configuración con la que empezamos pero rotada un ángulo recto. Esta regla muestra como con autómatas celulares pueden realizarse rotaciones de configuraciones con respecto a un eje de simetría.

9. `(v2 g d1 ia (!128!g:;) v0 g v2 g iw (!256!g:;) v1 g;)` Se realiza la misma acción que con la opción anterior, pero en este caso la diagonal se dibuja del extremo superior izquierdo al inferior derecho.

10. `(iV o4 (!256!g:;) v1 g;)` Aquí, se dibuja una línea horizontal en la pantalla y se produce un desplazamiento de la configuración y cada partícula se reflejará en la línea como si se tratara de un espejo.

11. `(iU V4 (!256!g:;) v1 g;)` Lo mismo que con la opción anterior salvo que ahora la línea es vertical.

12. `{(!255!g:;)c(oc9n@c;)a(Vc9e@c;)b(@a@b@a;;)}` Esta opción rota alguna configuración establecida en los planos, mediante desplazamientos parciales.

13. `{(!255!g:;)c(oc9s@c;)a(Vc9w@c;)b(@a@b@a;;)}` Es la inversa a la opción anterior, realizando la rotación mediante desplazamientos parciales en sentido contrario.

14. `(''CAMEX.PAT'' y1 ;;)` Esta opción permite cargar un patrón de un archivo, en vez de generarlo dentro del mismo programa.

15. `{(gk:;)a(zf n1 p2 iy @a H2 @ H1 @a:;)}` En esta demostración se pueden observar distintas regiones en el plano las cuales pueden pasar de un estado de "estabilidad" a uno aparentemente más "desordenado", y viceversa. El efecto visual es de células desplazándose por el espacio.

16. `(zf R v1 g R iv (gk:;);)` Establece la regla de Silverman, en la que varias células se desplazan en diferentes direcciones.

17. `(zf w1 "nds2wa. " a l (gk:;);)` En esta opción se dibujan varios deslizadores de Vida, y luego empiezan a desplazarse. Los deslizadores son dibujados mediante el uso de un operador llamado *pantograph*, que coloca células en estado uno en los Planos de Bits siguiendo instrucciones direccionales (hacia el sur, hacia el norte, en diagonal, etc.).

18. `(zf w1 "ns4e5n8w9s9e3e9n5n9w7w9s2e" a ib (gk:;);)` Dibuja espirales utilizando el operador *pantograph*, y en su interior evolucionan células que se desplazan por su superficie.

19. `zf h4 R iL g (gk:;);)` En esta demostración se colocan puntos dispersos en el plano, a partir de los cuales "crecen" líneas perpendiculares que forman una rejilla en los planos.

20. `(zf h4 R iL g (gk:;);)` Este es un Juego de la Vida en donde se corren cosas similares en los distintos planos, y causa un efecto atractivo con una paleta de colores adecuada.

3.7 Comentarios finales

La interfaz de CAMEX es muy útil para aquellos usuarios que simplemente están interesados en ver la forma en que se comportan los autómatas celulares pero que no están interesados (o familiarizados) con la programación a bajo nivel que sugiere el código abierto del programa. Las opciones que presenta el programa son útiles y permiten la realización de una gama bastante amplia de experimentos interesantes, permitiendo la interacción inmediata con la tarjeta CAM-PC. La mayoría de los comandos de consola de CAMEX son fáciles de aprender, y están asociados a operadores de REC, lo que da una confortable homogeneidad al programa en su totalidad.

CAMEXcuenta con muchas facilidades para una experimentacin muy flexible y variada con autómatas celulares, pero por la experiencia que hemos tenido al manipular la CAM-PC mediante CAMEX, sólo un pequeño subconjunto de comandos es el que realmente es en verdad útil y se usa recurrentemente. Ejemplos de éstos comandos son los que dibujan configuraciones aleatorias uniformes, elipses aleatorias, puntos esparcidos, correr la máquina, edición de programas en REC y el desplazamiento de planos.

Es necesario señalar la importancia del manejo de la interfaz de CAMEX en lo que toca a los parámetros de algunos de los operadores (comandos de consola), como la densidad de puntos dispersos en el plano, el número de alguna regla de Wolfram o el radio de la elipse aleatoria (que consideramos como una característica principal de CAMEX), ya que estos no pueden ser cambiados desde un programa de REC, por ejemplo.

El programa CAMEX fue desarrollado según los intereses de su autor, dependiendo de qué investigaciones realizaba. De estos intereses surgieron partes más o menos desarrolladas y posiblemente también se deba a esto el que existan opciones en el código que no se encuentran disponibles.

El lenguaje REC es en verdad algo muy diferente a lo que los planes de estudio de las carreras de computación nos tienen acostumbrados respecto a lo que es un lenguaje de programación, aún más que Forth. REC es, paradójicamente, más sencillo y a la vez más complicado que, por ejemplo, un lenguaje estructurado como Pascal o C. Lo que sucede es que cuando uno hace programas en el REC de la CAM-PC se cuenta con operadores que realizan diversas tareas y una estructura de control que es interpretada (y ejecutada en el momento) y que es muy flexible. Pero lo que ya no es tan sencillo para el usuario es lo que hay detrás de REC, que es código en C, y para crear o modificar subrutinas de REC se necesita programar en C y

tener conocimiento de la operación de la CAM-PC. Sin embargo este esfuerzo extra es el que nos proporciona todo el control que deseemos tener sobre la máquina.

En el siguiente capítulo se verá la forma en que el usuario puede crear sus propios operadores de REC, y queremos notar que prácticamente cualquier operador de REC puede ser asignado a alguna tecla específica, pero que en nuestro caso no se realizó ninguna modificación pues quisimos mantener intacta la interfaz creada por el doctor McIntosh.

Capítulo 4

Reprogramando CAMEX

En este capítulo nos introduciremos en el código fuente de CAMEX. Es aquí donde podemos aprovechar toda la flexibilidad que nos brinda el código abierto (*open source*) del programa. Con el código fuente a nuestra disposición, podremos libremente reescribir y recompilar CAMEX para hacerlo crecer según nuestras necesidades y gustos particulares.

En el documento [12] existe una buena cantidad de información técnica de CAMEX y es una referencia que puede ser de mucha utilidad para quien quiera profundizar sobre varios aspectos del programa. No obstante, en este documento no se explican muchas cosas del programa y contiene una cantidad considerable de información que no encaja con el programa CAMEX (como el manejo del ratón (*mouse*), la edición de bitplanos y reglas y algunas opciones del entorno, entre otras). Posiblemente esto se deba a que el documento se refiere a alguna versión más reciente de CAMEX que la que contamos.

En el Apéndice A.1 se pueden ver los archivos de los que consta el programa CAMEX y la manera de compilarlos.

Cada módulo del código fuente de CAMEX contiene información histórica, que incluye fechas de creación y revisión de los módulos.

Trataremos el tema de la reprogramación del programa CAMEX centrándonos principalmente en las vecindades, reglas y configuraciones de autómatas celulares, que es lo principal que debe saberse para aprovechar la tarjeta CAM-PC. Sin embargo CAMEX es un programa grande y hay otros aspectos que pueden ser explorados. El documento [12] contiene información sobre éstos aspectos del programa, entre los que se encuentran descripciones de reglas, tópicos de autómatas celulares unidimensionales, teoría de gráficas y autómatas celulares tridimensionales.

En este capítulo se muestra en principio la estructura de los diversos registros de la tarjeta CAM-PC que se utilizan en la programación de CAMEX. La explicación de cada registro puede resultar un tanto confusa y tal vez parezca críptica, pero recomendamos al lector no preocuparse por esto, ya que después de la explicación de los registros se ilustra la forma en que éstos se manipulan para la programación de configuraciones iniciales, vecindades y reglas. Al final del capítulo se ve como se pueden incluir en un operador de REC los programas que instalan los elementos de un automáta celular, lo cual permite llevar a cabo experimentos en CAMEX.

De este modo, la estructura de este capítulo ilustra primero los recursos de CAMEX que podemos aprovechar, después muestra la forma en que se pueden utilizar cada uno de los recursos mencionados por medio de un programa en lenguaje C, y finalmente se conjunta todo en un ejemplo que implanta la regla de Vida en CAMEX. Además, la última sección enseña los pasos a seguir para la inclusión de un nuevo operador REC en CAMEX.

4.1 Los registros de la CAM-PC

A lo largo de los capítulos anteriores se ha hablado de las características y los componentes principales de la CAM-PC, y también se mencionó que la comunicación entre la PC huésped y la tarjeta se hace a través de la Interfaz de la PC. Pues bien, dicha comunicación se realiza mediante la lectura y escritura de posiciones especiales de memoria o *registros* que están contenidos en la tarjeta. De hecho, este enfoque de 'mapeo de memoria' es el que usan comúnmente las tarjetas de circuitos que se añaden al *bus* de la PC, y en esta sección hablaremos de la función y uso de algunos registros de la CAM-PC.

4.1.1 Lectura y escritura de registros

Los registros pueden ser leídos o escritos como si fueran posiciones de memoria, de tal manera que se pueden utilizar las instrucciones ordinarias que en un lenguaje de programación permiten hacer referencia a la memoria. En el caso concreto de CAMEX dichas instrucciones son las propias de las versiones de lenguaje C de Borland, y deben ser usadas teniendo en cuenta las restricciones de lectura/escritura que aplican en cada uno de los diferentes registros de la CAM-PC.

En el presente trabajo sólo daremos la explicación de los registros que

sirven para los propósitos de establecimiento de vecindades, tablas de consulta y configuraciones en los autómatas celulares; si el lector desea conocer en su totalidad los registros, recomendamos revisar la tabla 4.2 del Manual de Hardware[13] distribuido por Automatrix junto con la tarjeta.

Volviendo a nuestro estudio, las operaciones de lectura y escritura se realizan mediante las instrucciones `peekb` y `pokeb`, respectivamente.

Nuestra atención la centraremos en la función `pokeb`, ya que para los fines perseguidos en este capítulo únicamente necesitaremos escribir en registros. La forma y los argumentos típicos de estas funciones (tomadas de la ayuda del la versión de lenguaje C *Borland C++ 5.0*) son:

pokeb

Sintaxis:

```
#include <dos.h>
void pokeb(unsigned segment, unsigned offset, char value);
```

Descripción:
Almacena el valor de un byte en la localidad de memoria *segment:offset.*

peekb

Sintaxis:

```
#include <dos.h>
char peekb(unsigned segment, unsigned offset);
```

Descripción:
Devuelve el valor del byte en memoria especificado por *segment:offset.*

Cada uno de los registros de la CAM-PC está formado por 8 bits etiquetados de 0 a 7, es decir, son *bytes*. Un bit de un registro está activo cuando contiene un 1, y está desactivado cuando tiene un valor de 0. Así, supongamos una situación en la que queremos activar sólo los bits 0 y 3 del registro AAR, es decir, buscamos la siguiente configuración de la Figura 4.1.

Para lograr esto simplemente debemos utilizar la función `pokeb` con los siguientes argumentos:

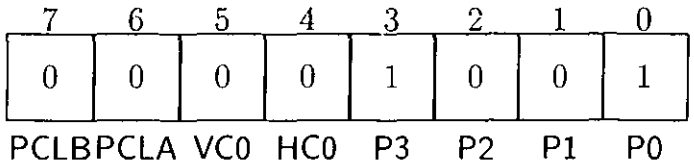


Figura 4.1: Configuración de bits en el registro AAR. El funcionamiento de este registro se explicará en la Sección 4.1.2.2.

```
pokeb (BASE, AAR, (char) 0x09)
```

donde:

- BASE indica a pokeb el segmento donde se halla la tarjeta CAM-PC;
- AAR indica la posición donde está ubicado el registro en que deseamos escribir; y
- (char) 0x09 es el valor que queremos escribir en el registro, expresado en notación hexadecimal para facilitar la lectura y escritura del programa[1] 0x09 es igual al número binario de ocho dígitos 00001001[2].

En este momento hay que señalar que para cada uno de los registros de CAM-PC existen constantes asociadas en el programa CAMEX, cuyos nombres corresponden al del registro físico que representan, y cuyo valor es un número hexadecimal igual al de la posición donde se localiza dicho registro en la tarjeta. Dichas constantes están declaradas en CAMPC.H. Lo anterior permite ver que el argumento AAR en la función pokeb no es el registro en sí, sino una constante con el valor 0x1FB (ver la Tabla 4.2 de [13]) , que corresponde a

[1]El uso principal para los números hexadecimales (así como los octales) en computación es para abreviar representaciones binarias largas. Los números binarios pueden ser expresados de forma concisa en sistemas numéricos con bases más altas que el binario. Luego, un hexadecimal se convierte fácilmente en un número binario tomando cada dígito y convirtiéndolo en un binario de 4 dígitos. En nuestro caso, 0=0000 y 9=1001; de este modo, 0x09=00001001.

[2]Las posiciones de los valores de cada bit deben ser tomadas como un número binario, y así, valores iguales a 1 en los bits 0 y 3 se convierten en $1x2^0$ y $1x2^3$ respectivamente, que al ser sumados dan como resultado el valor de 9 decimal, o 0x09 en la sintaxis de C para valores hexadecimales.

la posición de memoria que ocupa el registro AAR en la tarjeta. Esto ocurre en CAMEX para todos los registros de la tarjeta.

Para la identificar la tarjeta existe un valor similar, el cual está representado por la constante BASE.

4.1.2 El papel que juegan los registros

Hemos mencionado que la comunicación entre la PC huésped y la CAM-PC se hace mediante la lectura y escritura de registros, pero no hemos dado una explicación detallada del porqué es necesaria esta comunicación. Pues bien, es la comunicación entre ambas partes la que logra poner en marcha la simulación de un autómata celular: la CAM-PC tiene los recursos y la PC huésped le dice de que manera se utilizarán dichos recursos y cual será la información que se manejará.

A continuación presentaremos una descripción de los 8 registros que utilizaremos a lo largo de este capítulo. En la descripción que a continuación se presenta sólo se habla de lo concerniente a los registros y bits que sirven a nuestros propósitos.

4.1.2.1 El registro CCR

El nombre de CCR es un acrónimo para *Configuration and Control Register* (Registro de Configuración y Control). Su estructura se muestra en la siguiente figura:

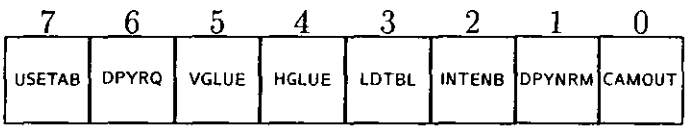


Figura 4.2: Estructura del registro CCR

Los bits de este registro manejan algunas de las características principales de la CAM-PC, que van desde definiciones de la fuente de video hasta la activación de los modos de operación. Sin embargo, a nosotros nos interesa solamente el bit número 3, LDTBL, cuyo valor permite definir la fuente de entrada de las primeras dos líneas de dirección de entrada de las tablas de

consulta[3]. De hecho, el valor de LDTBL siempre deberá ser igual a 0 para poder llevar a cabo el llenado de dichas tablas. El llenado de dichas tablas se realiza en conjunto con los registros TAA y TBA (ver Sección 4.1.2.3) y se explica en la Sección 4.3.3.2.

4.1.2.2 El registro AAR

Este es el *Auxiliary Address Register* (Registro Auxiliar de Direcciones), cuya función primaria es la de modificar las selecciones de los registros TAA y TBA, aunque también está involucrado en la importante operación de leer y escribir en el Buffer de Planos, y su configuración es:

7	6	5	4	3	2	1	0
PCLB	PCLA	VC0	HC0	P3	P2	P1	P0

Figura 4.3: Estructura del registro AAR

De este registro describiremos los bits 0, 1, 2, 3, 6 y 7:

Bit 7: PCLB Este bit, junto con el bit TBC1 en TBA, selecciona la fuente de entrada para los planos 2 y 3. Cuando TBC1=0, este bit puede usarse para cambiar entre las tablas de consulta auxiliares y primarias (ver Tabla 4.2).

Bit 6: PCLA Este bit, junto con el bit TAC1 en TAA, selecciona la fuente de entrada para los planos 0 y 1. Cuando TAC1=0, este bit puede usarse para cambiar entre las tablas de consulta auxiliares y primarias (ver Tabla 4.1).

Bits 3-0: P3-P0 Estos son llamados en ocasiones "bits de fase bajos" (*lower phase bits*). Cuando se cargan las tablas de consulta, P0, P1, P2 y P3 controlan las líneas de dirección de entrada 8, 9, 10 y 11 tanto en CAM-A como en CAM-B. Cuando se lee o se escribe en el Buffer de

[3]Las fuentes de entrada o líneas de dirección son los medios por los cuáles las tablas de consulta obtienen información sobre el estado de las células que conforman un vecindario (Ver Sección 2.2.1).

Planos, estos bits seleccionan los planos a los cuales se hace referencia (ver Tablas 4.3 y 4.4). Estos bits también pueden ser seleccionados como pseudovecinos (Sección 4.3.2) durante los pasos de la evolución.

Usar el bit:		Para tener la fuente de:	
TAC1	PCLA	Plano 0	Plano 1
0	0	PRI0	PRI1
0	1	AUX0	AUX1
1	0	PDAT	PDAT
1	1	UI0	UI1

Tabla 4.1: Combinaciones para seleccionar los datos de entrada del Buffer de Planos para CAM-A. Las combinaciones 10 y 11 no son utilizadas en el presente trabajo

Usar el bit:		Para tener la fuente de:	
TBC1	PCLB	Plano 2	Plano 3
0	0	PRI2	PRI3
0	1	AUX2	AUX3
1	0	PDAT	PDAT
1	1	UI2	UI3

Tabla 4.2: Combinaciones para seleccionar los datos de entrada del Buffer de Planos para CAM-B. Las combinaciones 10 y 11 no son utilizadas en el presente trabajo

4.1.2.3 Los registros TAA y TBA

Como se ha explicado en secciones anteriores, la CAM-PC cuenta con tablas de consulta que contienen las reglas que determinan la dinámica del autómata. A nivel de *hardware* las tablas involucran en sí dos operaciones: 1) hay que llenarlas, a fin de que contengan la regla del autómata, y 2) una vez llenas, y habiendo comenzado la simulación, es necesario presentar ante ellas la información correspondiente a la célula evaluada (su estado y el de sus

P1	P0	Plano leído
0	0	0
0	1	1
1	0	2
1	1	3

Tabla 4.3: Selección de los planos de bits para la operación de lectura

Bit de fase	Plano escrito
P0	0
P1	1
P2	2
P3	3

Tabla 4.4: Selección de los planos de bits para la operación de escritura

células vecinas). Ambas acciones se llevan a cabo mediante la adecuada habilitación de las líneas de dirección de entrada de las tablas.

De las acciones enunciadas en el párrafo anterior, la segunda puede ser plenamente identificada como la selección de las células que conforman el tipo de vecindario (Sección 2.2.2), y esto se logra utilizando los registros TAA y TBA, que controlan las líneas de dirección de entrada para CAM-A y CAM-B, respectivamente. En las tablas contenidas en esta sección las líneas de dirección de entrada serán identificadas con etiquetas de la forma TAAXX para CAM-A y TBAXX para CAM-B, donde XX indica el número de la línea de dirección de entrada con que se está tratando. Por ejemplo, la línea de dirección de entrada 0 de CAM-A será TAA00, y la cuarta línea de dirección de entrada de CAM-B será TBA03.

7	6	5	4	3	2	1	0
TAC1	TAM	TAS5	TAS4	TAS3	TAS2	TAS1	TAS0

Figura 4.4: Estructura del registro TAA

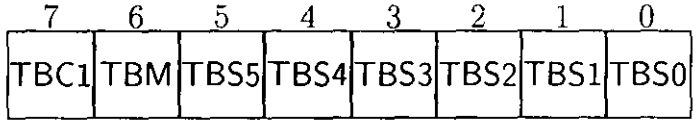


Figura 4.5: Estructura del registro TBA

Ambos registros trabajan de manera idéntica (en la mitad de CAM que le corresponde a cada uno), y su descripción es la siguiente:

Bit 7: TAC1/TBC1 Este bit se combina con el bit PCLA/PCLB del registro AAR para cambiar entre las tablas primarias y las auxiliares.

Bit 6: TAM/TBM Este bit es ocupado para establecer la vecindad de Margolus cuando su valor es igual a 1. Si el valor del bit es 0, entonces se permite el uso de las vecindades de Moore y von Neumann. En el presente trabajo sólo se trata con experimentos que ocupan las dos últimas vecindades mencionadas, si el lector desea conocer la forma en que trabaja la vecindad de Margolus le recomendamos consultar el capítulo 12 de [17] y el capítulo 11 de [12].

Bits 5-0: TAS5/TBS5 a TAS0/TBS0 Estos bits se activan en diversos órdenes y sus combinaciones permiten hacer corresponder las líneas de dirección de entrada de las tablas con las vecindades preconstruidas de CAM (en las Tablas 4.5, 4.6, 4.7, 4.8 y 4.9 se puede ver la forma en que las combinaciones de valores en estos bits determinan las células que conforman una vecindad). Por ejemplo, al asignar a TAS0 el valor de 0, a TAS1 el valor de 1, a TAS 2 el valor de 1, y a TAS3 el valor de 0 (combinados con TAM=0 y LDTBL=0) obtenemos la vecindad de Moore en CAM-A, es decir, tenemos como miembros de la vecindad a las líneas de entrada que representan a los vecinos noroeste (NW0), noreste (NE0), suroeste (SW0), sureste (SE0), norte (N0), sur (S0), oeste (W0) y este (E0) del Plano 0, junto con el centro de este mismo plano (C0) y el centro del Plano 1 (C1). Para la vecindad de von Neumann en CAM-B los valores de los bits deben ser TBS0=1, TBS1=0, TBS2=1 y TBS3=0 (combinados con TBM=0 y LDTBL=0), obteniendo a un vecindario conformado por el este (E3), oeste (W3), sur (S3) y norte

(N3) del Plano 3, y el este (E2), oeste (W2), sur (S2) y norte (N2) del Plano 2, aparte de los centros de ambos planos (C3 y C2).

LDTBL	TBA01	TBA00	TAA01	TAA00
0	C3	C2	C1	C0

Tabla 4.5: Selección de las líneas de dirección de entrada para los centros de cada plano. En general, sólo se contempla el caso de LDTBL=0 porque sirve para todas las vecindades. El caso en que se utiliza este bit con valor igual a 1, simplemente selecciona las líneas de dirección de entrada más bajas (0 y 1), para lo cual no se menciona ningún uso específico en los textos consultados por nosotros.

TAS1	TAS0	TAA05	TAA04	TAA03	TAA02
0	1	N1	S1	W1	E1
1	0	NW0	NE0	SW0	SE0

TAS3	TAS2	TAA09	TAA08	TAA07	TAA06
0	1	N0	S0	W0	E0
1	0	V0	H0	V1	H1

Tabla 4.6: Selección de las líneas de dirección para las vecindades de Moore y von Neumann en CAM-A. El bit TAM debe ser igual a 0. También se deben combinar con LDTBL=0 (ver tabla 4.5)

4.1.2.4 El registro TDAT

La acción de llenado de las tablas de consulta se logra mediante el uso de los registros AAR y TDAT. El primero ya ha sido explicado, así que toca el turno a TDAT.

TDAT es el registro en el que los datos de las tablas son escritos o leídos. Este registro tiene una estructura diferente de los que hemos revisado anteriormente, puesto que éste en lugar de ocupar un solo byte ocupa 256 bytes, cada uno de los cuales ayuda a definir las combinaciones que se pueden

TBS1	TBS0	TBA05	TBA04	TBA03	TBA02
0	1	N3	S3	W3	E3
1	0	NW2	NE2	SW2	SE2

TBS3	TBS2	TBA09	TBA08	TBA07	TBA06
0	1	N2	S2	W2	E2
1	0	V0	H0	V1	H1

Tabla 4.7: Selección de las líneas de dirección para las vecindades de Moore y von Neumann en CAM-B. El bit TBM debe ser igual a 0. También se deben combinar con LDTBL=0 (ver tabla 4.5)

TAS5	TAS4	TAA11	TAA10
0	0	P3	P2
0	1	C3	C2
1	0	V0	H0
1	1	UA11	UA10

Tabla 4.8: Selección de las líneas de dirección para las pseudovecindades en CAM-A (Sección 4.3.2)

TB5	TBS4	TBA11	TBA10
0	0	P3	P2
0	1	C1	C0
1	0	V0	H0
1	1	UB11	UB10

Tabla 4.9: Selección de las líneas de dirección para las pseudovecindades en CAM-B (Sección 4.3.2)

dar en las primeras 8 líneas de dirección de entrada de las tablas, complementándose con las líneas de dirección de entrada que se seleccionan con AAR (Sección 4.1.2.2).

CAM cuenta con 8 tablas de consulta, 4 tablas principales y 4 auxiliares, que están relacionadas con cada uno de los planos (ver Sección 2.2.1), las cuales pueden ser leídas o escritas al mismo tiempo, incrementando secuencialmente desde cero la parte de la dirección que corresponde a los bits de fase (seleccionados en AAR y leyendo o escribiendo en TDAT), en transferencias de 256 bytes.

La estructura de TDAT se muestra en la Figura 4.6.

7	6	5	4	3	2	1	0
AUX3	PRI3	AUX2	PRI2	AUX1	PRI1	AUX0	PRI0

Figura 4.6: Estructura del registro TDAT

Cada bit del registro corresponde a cada una de las tablas de consulta, y se puede ver que en el mismo momento se escribe en la misma posición de cada tabla.

Como se mencionó en secciones anteriores, las combinaciones de los bits TAC1 y TBC1, junto con la selección de PCLA y PCLB en AAR (ver Tablas 4.1 y 4.2), permiten cambiar entre las reglas contenidas en las tablas primarias y las reglas contenidas en las tablas auxiliares de CAM-A y CAM-B. Por ejemplo, si quisiéramos tener como fuentes para la transición del autómata celular a las tablas primarias CAM-A, sin importar si la vecindad es de Moore o von Neumann, la combinación de los registros AAR y TAA está mostrada en las Figuras 4.7 y 4.8.

4.1.2.5 Los registros PCA, PRA y PDAT

Estos tres registros se explican juntos en esta sección debido a que están estrechamente ligados. La función que cumplen es permitir las operaciones de lectura y escritura en el Buffer de Planos.

El Buffer de Planos almacena la información correspondiente al estado de las células de cada plano. Como hemos mencionado, el Buffer de Planos contiene 4 planos de bits, cada uno compuesto por 256 renglones y 256 columnas,

7	6	5	4	3	2	1	0
0	0	0 ó 1	0 ó 1	0 ó 1	0 ó 1	0 ó 1	0 ó 1
TAC1	TAM	TAS5	TAS4	TAS3	TAS2	TAS1	TAS0

Figura 4.7: Configuración de bits en el registro TAA que establece como fuente de entrada para la transición a las tablas primarias en CAM-A.

7	6	5	4	3	2	1	0
0	0	0 ó 1	0 ó 1	0 ó 1	0 ó 1	0 ó 1	0 ó 1
PCLB	PCLA	VC0	HC0	P3	P2	P1	P0

Figura 4.8: Configuración de bits en el registro AAR que establece como fuente de entrada para la transición a las tablas primarias en CAM-A, al mezclarse con el registro TAA. Para nosotros los bits 4 y 5 pueden tener cualquier valor pues no los usamos dentro de este trabajo.

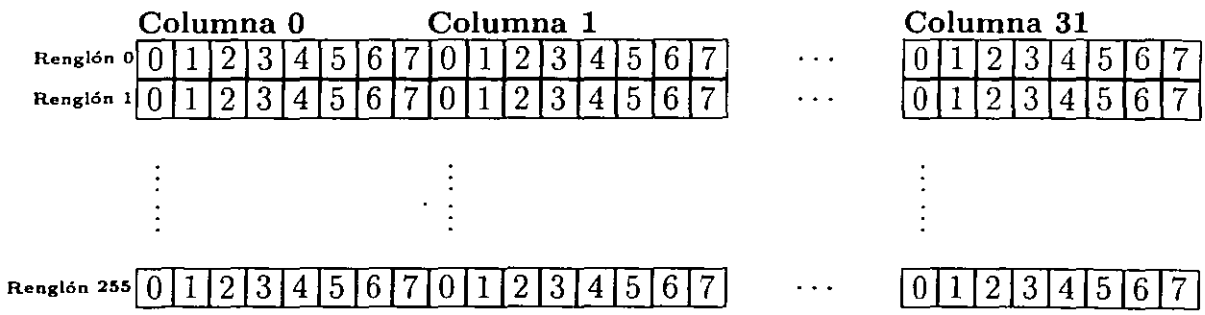


Figura 4.9: Estructura de un Plano de bits, tomando en cuenta la forma en que trabajan los registros PRA y PCA. Puede verse que las columnas están compuestas por 8 bits cada una.

que el Procesador CAM lee y escribe un lugar a la vez. Sin embargo, desde el punto de vista del programador y de la PC huésped, los datos se leen y se escriben byte por byte de algún plano de bits en particular. Los datos en cada plano se almacenan horizontalmente con el bit menos significativo en el lado izquierdo —contrario al almacenamiento numérico—. De este modo, los datos contenidos en el Buffer de Planos son referidos seleccionando un plano de bits, el índice del renglón con un valor entre 0 y 255, y el índice de un byte[4] con un valor entre 0 y 31 (ver Figura 4.9).

La selección de los planos de bits, tanto para lectura como para escritura, se vió al describir el registro AAR. Ahora bien, la selección del renglón se hace mediante el registro PRA (*Plane Row Address*), el cual acepta valores numéricos entre 0 y 255, por lo que no es necesario hacer ninguna distinción especial sobre la función de cada bit (ver Figura 4.10).

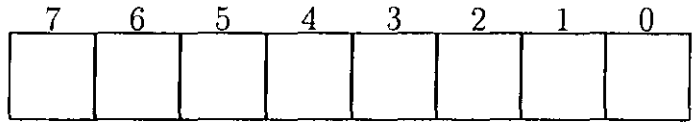


Figura 4.10: Estructura del registro PRA

El registro PCA (*Plane Column Address*)trabaja de manera parecida a PRA, y nos permite seleccionar el byte que deseamos leer o escribir (ver Figura 4.11).

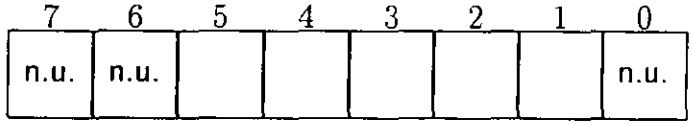


Figura 4.11: Estructura del registro PCA (n.u.=no utilizado)

Habiendo explicado que la referencia para operar con los planos de bits se hace por bytes, y que por tanto nuestro número de columnas será de

[4]Cada byte contiene en sí los índices de 8 columnas, lo que complica un poco la escritura en los planos (ver Sección 4.2.1.3).

$256 \div 8 = 32$, es obvio que sólo necesitaremos 5 bits para obtener los 32 índices que requerimos. Así, en la Figura 4.11 podemos ver que PCA sólo tiene 5 bits disponibles, recorridos 1 bit a la izquierda[5]. De este modo, este registro acepta valores entre 0 y 63, ignorando el valor del bit menos significativo, lo cual nos lleva a la obtención de sólo los valores pares entre 0 y 63 (incluyendo al cero), que constituirán nuestros 32 índices. De este modo, para escoger la tercera columna nuestro índice deberá ser igual a 4, el cual se escribiría mediante la siguiente instrucción:

```
pokeb (BASE, PCA, (char) 0x04)
```

Obtendríamos el mismo resultado si escribiéramos el valor de 5, ya que el primer bit de PCA se ignora. El programador debe tener esto presente para eliminar la posibilidad de escribir dos veces la misma columna.

Una vez seleccionado el plano correcto, así como el renglón y la columna precisos, ocuparemos el registro PDAT el cual contendrá los datos que se van a leer o a escribir. Este registro presenta la siguiente forma:

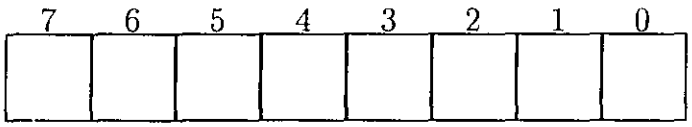


Figura 4.12: Estructura del registro PDAT

Dado que PDAT tiene una longitud de 8 bits, el valor que contenga llenará cada una de las 8 columnas representadas por los bits del registro PCA.

Por ejemplo, para escribir 8 bits con valor igual a 1 en las columnas 16 a 23 (tercera columna de PCA) en el segundo renglón (renglón 1 de PRA), el registro PDAT debe contener el valor decimal 255, que llenará de unos el registro, mientras que el valor de PCA debe ser igual a 6, y el de PRA debe ser igual a 1. En las Secciones 4.2.1.2, 4.2.1.3 y 4.2.1.4 se muestra cómo hacer programar esto en CAMEX.

[5]En [13] se menciona que dicho corrimiento se da "por razones históricas", de las cuales no encontramos explicación en ningún documento relacionado con la CAM-PC.

4.2 Establecimiento de configuraciones iniciales

En las simulaciones de autómatas celulares es de gran importancia la definición de una configuración inicial de estados del autómata, a fin de observar la manera en que actúa realmente la dinámica impuesta por las reglas. Las configuraciones pueden ser patrones ordenados de acuerdo a los gustos del experimentador, o simplemente pueden estar formadas por células en estados arbitrarios esparcidas de manera aleatoria a lo largo de los planos o de alguna sección específica de ellos.

El establecimiento de configuraciones iniciales se logra a través de la escritura en el Buffer de Planos, valiéndonos de los registros de la CAM-PC.

4.2.1 Manejo del Buffer de Planos

El Buffer de Planos almacena tanto las configuraciones iniciales del autómata como los cambios que se presentan en dicha configuración durante la evolución del autómata. En el Buffer de Planos pueden realizar operaciones de lectura y escritura el Procesador CAM y la PC huésped. En la sección anterior mencionamos que la lectura y escritura del Buffer de Planos es realizada de forma secuencial por el Procesador CAM encargándose de un bit a la vez, mientras que desde la PC huésped dicho proceso se realiza byte por byte, lo que complica un poco la realización de estas operaciones para el programador. En cualquier caso, se hace necesaria una explicación referente al uso de los registros de la CAM-PC que nos permitirán establecer configuraciones, recordando que esto implica la operación de escritura, para lo cual hay que determinar primero el plano en que vamos a escribir, seleccionar un renglón y una columna, y finalmente determinar los bits que conformarán la configuración. También se podría seleccionar primero la columna y a continuación el renglón, aunque este orden representa una complejidad mayor para el programador, pues hay que recordar que las columnas no pueden ser elegidas directamente debido a la estructura del registro PCA. En el ejemplo ilustrativo al final de esta sección se verá como en realidad es más conveniente elegir primero el renglón y después la columna.

4.2.1.1 Selección del plano

La selección del plano de bits se hace utilizando el registro AAR, con las combinaciones apropiadas de los bits P0, P1, P2 y P3 (ver Tabla 4.4). Por

la manera en que se manejan los bits mencionados, es posible escribir en más de un plano a la vez. En concreto, si deseamos escribir en el plano 0 utilizaremos la siguiente instrucción:

```
pokeb(BASE, AAR, (char) 1)
```

la cual provoca que AAR muestre la siguiente configuración:

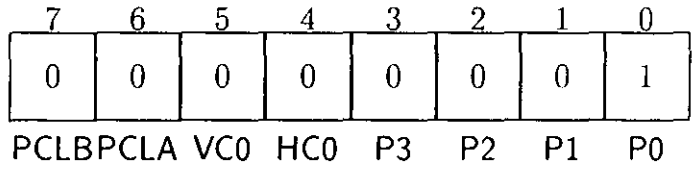


Figura 4.13: Configuración de bits en el registro AAR

Si quisiéramos escribir, por decir, en el Plano 3, el tercer argumento de `pokeb` debería ser 8, ya que $2^3 = 8$. Algo importante que es necesario resaltar es el hecho de que el tercer argumento de `pokeb` es un entero decimal forzado[6] a convertirse en carácter, para así cumplir con el prototipo de la función.

Por último, presentamos aquí una tabla que muestra los diversos valores que puede contener AAR a fin de escribir en el o los planos deseados.

4.2.1.2 Selección del renglón

La selección del renglón es bastante directa, ya que existen 256 renglones y el registro PRA maneja valores entre 0 y 255. Luego, si deseamos escribir en el renglón 115 simplemente usamos la siguiente instrucción:

```
pokeb(BASE, PRA, (char) 115)
```

La línea de código anterior podría repetirse una y otra vez, y la selección siempre sería el renglón 115. Sin embargo, cuando comienza la ejecución de la simulación de un autómata celular el registro cambia y se desfasa, por lo que en la programación de cualquier experimento se debe tener la precaución

[6] A esta operación se le llama "*cast*" en el lenguaje C, y en nuestro caso particular utilizamos la instrucción `(char)`.

Hexadecimal	Decimal	Planos Escritos
0	0	-
1	1	0
2	2	1
3	3	0,1
4	4	2
5	5	0,2
6	6	1,2
7	7	0,1,2
8	8	3
9	9	0,3
a	10	1,3
b	11	0,1,3
c	12	2,3
d	13	0,2,3
e	14	1,2,3
f	15	0,1,2,3

Tabla 4.10: Valores de AAR que permiten la escritura única o múltiple de los planos de bits

de reiniciar el registro[7] y obtener de esta manera el resultado deseado al establecer una configuración. Así, la sintaxis correcta para escoger un renglón es

```
pokeb(BASE, PRA, (char) rorg + 115)
```

donde `rorg` es una variable global de CAMEX (definida en `CAM.C`) con valor igual a 256. En otras palabras, `rorg` evita el desfase del que hablamos anteriormente, ya que el valor que se introduce en el registro es de 371, lo que desborda al registro y asegura que éste tenga almacenado el valor de 115.

4.2.1.3 Selección de la columna

En secciones anteriores se explicó el funcionamiento del registro PCA, por lo que sabemos que la selección de una columna implica cierta complejidad aritmética. Entonces, si la columna que deseamos escoger es la 27, debemos escribir:

```
pokeb(BASE, PCA, (char) corg + 54)
```

donde `corg`=256 y su valor está definido en el módulo `CAM.C`(este registro también se desfasa y hay que tomar las mismas precauciones que se tomaron con `PRA`). De manera general, el número de la columna que necesitamos debe ser multiplicado por 2, lo que da como prototipo de selección de columna la instrucción:

```
pokeb(BASE, PCA, (char) corg + 2*j)
```

donde `j` es el número de columna deseado.

4.2.1.4 Escritura de bits en los planos

El paso final en el establecimiento de una configuración es la determinación del o los bits que la conformarán. Debido al hecho de que la escritura se realiza byte por byte es necesario conocer de antemano la posición exacta de los bits que tendrán valor igual a 0 y de los que tendrán valor igual a

[7]Esto debe hacerse porque el registro se queda con el valor del último renglón seleccionado, y si el registro no es limpiado, se comienza la escritura de planos y el dibujado de pantalla partiendo desde este valor —de allí el desfase.

1. Por ejemplo, en el caso de que quisiéramos colocar una sola célula en estado 1 tendríamos hasta 8 bytes diferentes para escribir, siendo dos de estas posibilidades, por decir, los bytes 01000000 y 00001000. La escritura de estos bytes tendría la siguiente forma (recordando que el almacenamiento en los planos se realiza de forma inversa al almacenamiento numérico):

```
pokeb(BASE, PDAT, (char) 2) -para 01000000-
pokeb(BASE, PDAT, (char) 8) -para 00001000-
```

La escritura de configuraciones presenta casos triviales como el llenar algún plano con ceros o el llenarlo con unos, y por otro lado también se presentan casos con cierta dificultad en los que podemos utilizar los operadores de corrimiento de bits del lenguaje C, como se verá en el siguiente ejemplo ilustrativo.

4.2.1.5 Ejemplo ilustrativo: la función `onedot`

La función `onedot` se halla en el archivo `CAMPL.C` y su código es el siguiente:

```
/* a single dot at coordinates (c,r) in a field of zeroes in plane p. */
onedot(p,r,c) int p, r, c; {int i, j;
  /* Funciones para entrar al modo ocioso de operacion */
  wait();
  pnotv();
  /* Seleccion del o los planos usando el argumento 'p' */
  pokeb(BASE,AAR,(char)p);
  /* Ciclo para recorrer los renglones del o los planos seleccionados*/
  for (i=0; i<CL; i++) {
  /* Aqui se escoge el renglon */
    pokeb(BASE,PRA,rorg+i);
  /* Ciclo para recorrer las columnas del o los planos seleccionados */
    for (j=0; j<32; j++) {
  /* Aqui se escoge la columna */
      pokeb(BASE,PCA,corg+2*j);
  /* Condicion para decidir la escritura en el o los planos */
      pokeb(BASE,PDAT,(char)(i==r)?((j==c/8)?(0x80)<<(c%8):0):0);
      }
    }
  /* Funcion que reanuda el despliegue en pantalla */
  gotv();
}
```

Como lo indica el comentario, esta función coloca un solo punto en las coordenadas indicadas `c` y `r` en el plano `p`. Las instrucciones en lenguaje C manejan directamente los registros de la tarjeta CAM-PC, y así, en el análisis línea por línea nos encontramos la siguiente interpretación:

- Las funciones `wait()` y `pnotv()` forman parte de CAMEX, y sirven para detener la ejecución del modo de despliegue o el de procesamiento de la CAM-PC, debido a que las transferencias largas de información deben ser hechas en modo ocioso. La primera simplemente pone a trabajar funciones que no tienen una sola instrucción, mientras que la segunda provoca la suspensión del dibujado de la pantalla mientras se cargan los planos de bits. Similar a estas funciones es la función `gotv()`, la cual se halla al final del código y cuya ejecución reanuda el despliegue en pantalla.

- La instrucción `pokeb(BASE,AAR,(char)p)` selecciona el bit correspondiente a alguno de los planos de bits en el registro AAR. Se puede escribir en los cuatro planos al mismo tiempo, suministrando algún valor hexadecimal entre 1 y `f`, como se muestra en la Tabla 4.10.

- El ciclo `for` que contiene a `i` va desde 0 hasta 255, ya que CL está definido como igual a 256. Este ciclo servirá para indicar el renglón del plano en que se va a escribir, en la llamada a `pokeb` que le sigue inmediatamente.

- El siguiente ciclo, con índice `j`, ayuda a seleccionar la columna del plano en la cual deseamos escribir, al ir tomando valores enteros desde 0 hasta 31 que se utilizan como parte del argumento de la llamada a `pokeb` que trabaja con PCA.

- Dentro de los dos ciclos anidados que tenemos se encuentra otra llamada a la función `pokeb`, y ésta coloca en el registro PDAT el valor del byte que queremos escribir en el plano seleccionado. Ahora bien, el valor que se coloca en PDAT está dado por las expresiones condicionales contenidas en el tercer argumento de `pokeb`, que es

  ```
  (i==r)?((j==c/8)?(0x01)<<(c%8):0):0
  ```

  La primera condición evalúa el valor del índice `i` para saber si nos encontramos en el renglón en que deseamos colocar el punto; si el renglón

no es el deseado simplemente colocamos un byte que contiene solamente ceros en el plano, pero si hemos encontrado nuestro renglón la condición nos conduce a la búsqueda de la columna deseada, la cual deberá ser dividida entre 8 para poder ser comparada con el índice j. Nuevamente la condición nos indica colocar un byte con ceros en el plano si no nos hallamos en la columna que deseamos, en caso contrario colocaremos un uno en el plano, aunque esta operación no es tan directa como nos podría parecer ya que hay que recordar que cada una de las columnas indicadas por el registro PCA contiene en realidad 8 bits, los cuales son las columnas reales del plano. Así, para conseguir esto hay que obtener el residuo de la división entera de la columna deseada entre 8, y así tendremos el valor del bit exacto al que hay que desplazarnos (dentro de la columna suministrada por PCA) para colocar nuestro punto. Obviamente hay que tener primero un bit en la posición c÷8 (columna exacta de acuerdo con PCA) y después recorrerlo c%8 lugares usando el operador de C de desplazamiento de bits a la izquierda. En forma extensa, el valor hexadecimal 0x01 (1 en base 10) está representado por el siguiente byte:

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1

Si quisiéramos escribir un 1 exactamente en la columna número 13, lo que haríamos sería ir a la columna que PCA reconoce como la columna 1 (recordemos que cada columna a que se refiere PCA contiene en realidad 8 columnas del plano, y que entonces en la columna 1 de PCA están las columnas 8, 9, 10, 11, 12, 13, 14 y 15 del plano). Si escribiéramos directamente el valor 0x01 en esta columna, lo que tendríamos sería un 1 en la columna 8. Es por esto que para colocarnos en la columna del plano deseada debemos desplazar nuestro bit (es decir, el uno) 5 lugares a la izquierda (5 es el residuo de la división entera de 13 entre 8). Luego, el byte que escribiríamos es el siguiente:

Por último hay que señalar que las operaciones realizadas hasta el momento son las correctas aunque parezca lo contrario, pues hay que recordar que el almacenamiento en el Buffer de Planos se realiza de manera

7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0

inversa al almacenamiento numérico, es decir, el bit menos significativo se halla en el extremo izquierdo (ver Sección 4.1.2.5). Así, el byte que se escribirá en el plano es el siguiente:

8	9	10	11	12	13	14	15
0	0	0	0	0	1	0	0

4.3 Vecindades y tablas de consulta

En la Sección 2.2.2 se vio cómo la CAM soporta vecindades desde el nivel del *hardware*. Aquí veremos como se manipulan desde el programa CAMEX las vecindades.

Las vecindades y las tablas de consulta son los últimos componentes esenciales para el manejo de autómatas celulares en la CAM-PC. Tanto las tablas como las vecindades están contenidas en el Procesador CAM, y la relación que existe entre ellas es bastante estrecha dado que las vecindades contienen la información que ocuparán las tablas para poder transformar el estado actual de las células que están siendo evaluadas, y así llevar a cabo la evolución del autómata.

4.3.1 Estructura de las tablas de consulta

Como se ha mencionado anteriormente, la CAM-PC tiene en total 8 tablas de consulta, de las cuales 4 corresponden a CAM-A y 4 a CAM-B (ver Figura 4.14). En ambos casos las tablas cuentan con 16 líneas de dirección de entrada que son habilitadas mediante el uso de algunos registros de CAM. Sin embargo, tanto el *software* original de la CAM-PC como CAMEX utilizan sólo 12 de

estas líneas de dirección. Cada tabla está compuesta por registros que corresponden a cada una de las posibles combinaciones que se pueden dar por la información que suministran las líneas de dirección de entrada, y dado que esta información no es más que el estado de un bit, el total de registros es de 2^{16}. En nuestro caso, y para ser congruentes con el *software*, consideremos que las tablas cuentan en realidad con sólo $2^{12} = 4096$ registros.

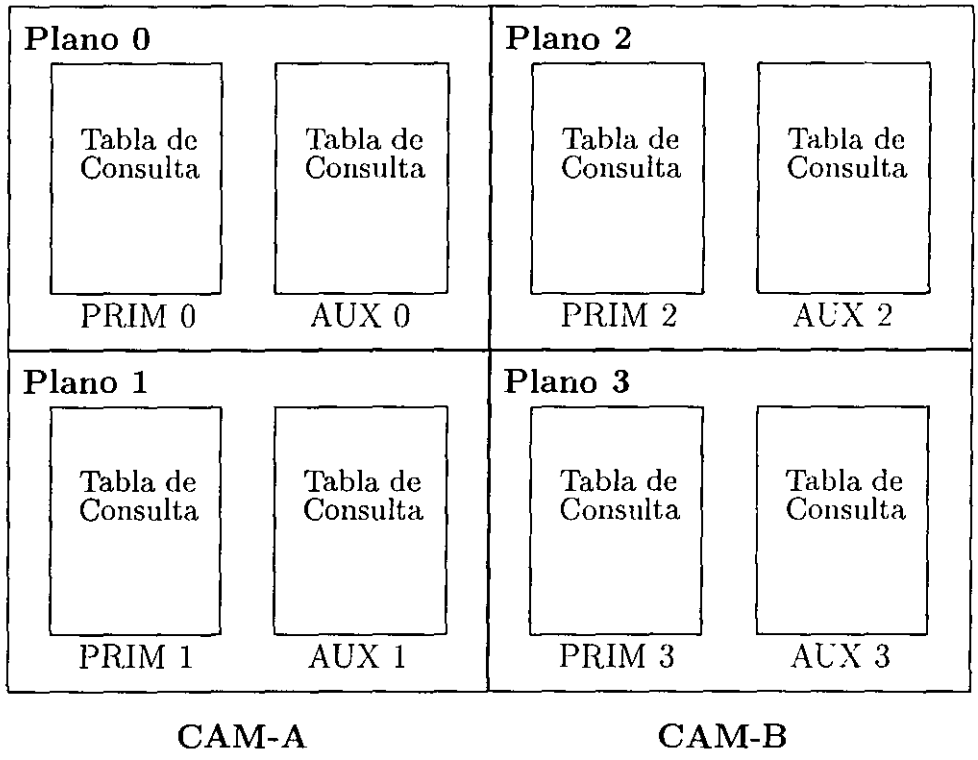


Figura 4.14: A cada mitad de CAM le corresponden 4 tablas de consulta.

Para llenar las tablas necesitamos entonces habilitar sólo las primeras 12 líneas de dirección de entrada, lo cual se logra a través de la escritura de los registros AAR y TDAT (ver Sección 4.1.2.4).

Hasta donde se ha mencionado, la habilitación de las líneas de dirección de entrada sólo se ha realizado para el llenado de las tablas. Una vez llenas las tablas, las líneas de dirección de entrada proporcionarán la información correspondiente al estado del vecindario de la célula evaluada, mediante la manipulación de algunos registros, lo cual será explicado a lo largo de esta sección. Sin embargo, al ser llenadas las tablas el programador debe tener presente la vecindad que usará en su experimento, puesto que la asociación de

vecinos a las líneas de dirección de entrada varía completamente de un tipo de vecindad a otra. En otras palabras, si el experimentador no tiene el suficiente cuidado podría llenar las tablas de consulta con una regla que utilice vecindad de Moore (como Vida) y combinarla con una vecindad diferente (como la de von Neumann), obteniendo resultados que sean totalmente diferentes a los esperados.

4.3.2 Vecindades preconstruidas en la CAM-PC

Una de las principales características de la CAM-PC es que cuenta con vecindades preconstruidas, es decir, combinaciones obtenidas al escribir en los registros TAA, TBA y AAR que indican qué línea de dirección de entrada va a proveer cuál información del vecindario a las tablas de consulta. Estas vecindades preconstruidas son en sí más que suficientes para experimentar con diversos autómatas celulares, pues son de las más populares: vecindad de Moore y vecindad de von Neumann. En adición a estas dos también se halla la vecindad de Margolus, ideada por Norman Margolus (ver el Capítulo 12 de [17]).

Las vecindades anteriores dejan libres algunas líneas de dirección de entrada, ocupando a lo más 10 de las 12 que existen, y se les llama "asignaciones mayores". Las líneas de dirección ocupadas por estas asignaciones mayores llevan información que se refiere estrictamente a los componentes del vecindario. Por otro lado, de las líneas sobrantes la CAM-PC utiliza 2 para el establecimiento de lo que se conoce como asignaciones menores o pseudovecindarios. Este último nombre fue acuñado a partir del hecho de que las líneas de dirección de entrada asignadas a éstos no siempre contienen información sobre el estado de una célula en el plano, sino que en ocasiones proveen información espacial o temporal. En total las pseudovecindades son 4:

CENTERS Conecta dos líneas de dirección a la otra mitad de CAM, dándonos una ventana de 1x1 en los otros 2 planos. En sí, esta pseudovecindad permite ocupar la información sobre el estado de las células centrales de la otra mitad de CAM.

PHASES Conecta dos líneas de dirección que proveen información temporal para poder trabajar con reglas distintas en pasos diferentes de la evolución. Para trabajar con este pseudovecindario es necesario manipular los bits P2 y P3 de AAR, y aparte es necesario crear un ciclo que

nos brinde información sobre los pasos de evolución del autómata. Esta información debe ser un bit, y entonces lo que nos diría esta vecindad es si el paso actual es par o impar al alternar entre valores de 0 y 1, o podríamos hacer algo más complejo si mezclamos los valores de ambos bits de fase, teniendo entonces hasta 4 diferentes etiquetas para los pasos de la evolución del autómata. Una regla que puede aprovechar esta pseudovecindad es la de Borde/Hueco (ver Sección 1.7.6).

HV Provee dos pseudovecinos en dos líneas de dirección que dan información sobre el espacio de evolución del autómata. La `H` es la fase horizontal y la `V` es la fase vertical, y funcionan proporcionando un índice de paridad (valores de 0 y 1) a cada renglón y a cada columna del plano, para de este modo establecer reglas de evolución que se refieran a la posición de la célula evaluada.

USER En esta pseudovecindad las líneas de dirección sobrantes se hallan asignadas a dos *patas* del conector del usuario (*user connector*).

En las tablas 4.11 y 4.12, incluidas a continuación, se muestra la manera en que son asignadas las líneas de dirección de entrada para cada una de las vecindades preconstruidas, tanto asignaciones mayores y menores, indicando las células de ambas mitades de CAM, CAM-A/CAM-B:

Línea de dirección	Moore	V. Neumann	Identificador CAMEX
0	Centro 0/2	Centro 0/2	a0
1	Centro 1/3	Centro 1/3	a1
2	Sureste 0/2	Este 1/3	a2
3	Suroeste 0/2	Oeste 1/3	a3
4	Noreste 0/2	Sur 1/3	a4
5	Noroeste 0/2	Norte 1/3	a5
6	Este 0/2	Este 0/2	a6
7	Oeste 0/2	Oeste 0/2	a7
8	Sur 0/2	Sur 0/2	p0
9	Norte 0/2	Norte 0/2	p1

Tabla 4.11: Vecindades preconstruidas en CAM-PC (asignaciones mayores)

En las Figuras 4.15 y 4.16 se ve la disposición de los vecinos de acuerdo a los datos de la Tabla 4.11. Del mismo modo se puede ver que la vecindad de

Línea de dirección	CENTERS	PHASES	HV	USER	Identificador CAMEX
10	Centro 2/0	Fase 0	H	UA10	p2
11	Centro 3/1	Fase 1	V	UA11	p3

Tabla 4.12: Pseudovecindades preconstruidas en CAM-PC (Asignaciones menores)

von Neumann tiene el mismo número de vecinos en cada plano, permitiendo el manejo de hasta 4 estados diferentes, que pueden ser tomados indistintamente para hacer una cuenta de vecinos que influya en la dinámica del autómata. Por su parte, la vecindad de Moore cuenta con 9 vecinos en su primer plano y sólo puede aprovechar un vecino en el segundo, por lo que si se toman 4 estados diferentes la transición de algunos de ellos será directa, sin hacer cuenta de vecinos; o tal vez se pueda usar ese único vecino en el segundo plano para ser el "eco" o el "rastro" (ver Sección 2.2.1) del primero, es decir, en el segundo plano se registra la historia del primero. Todo esto se debe al hecho de que no hay manera de "ver" el estado de las células que se hallan encima de los vecinos ubicados alrededor del centro. Para poder hacer esto, CAM necesitaría tener al menos 18 líneas de dirección de entrada. En otras palabras, la limitación del *hardware* es la causa de que no se pueda tomar en cuenta el estado de las células no centrales en los planos 1 y 3.

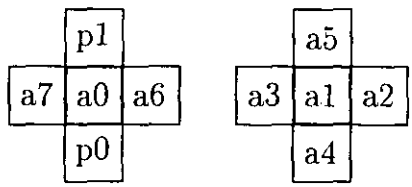


Figura 4.15: Disposición de los vecinos en CAMEX para la vecindad de von Neumann

Para poder ocupar cualquiera de las vecindades preconstruidas basta con escribir en los registros TAA y/o TBA (que se verán en la siguiente sección), siguiendo las convenciones de las Tablas 4.6 y 4.7. Por ejemplo, si quisiéramos

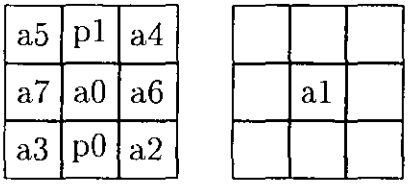


Figura 4.16: Disposición de los vecinos en CAMEX para la vecindad de Moore

tener la vecindad de von Neumann en CAM-A junto con la pseudovecindad de CENTERS, la escritura en TAA sería la siguiente:

```
pokeb(BASE, TAA, (char) 0x15)
```

Antes de pasar a la siguiente sección es necesario dejar en claro algunos puntos en lo que toca a las vecindades: aunque la vecindad de Margolus es una de las principales características de la CAM-PC en este trabajo no la estudiaremos, sin embargo, es conveniente mencionar que CAMEX cuenta con provisiones para su uso. Si se desea trabajar con esta vecindad, el experimentador debe conocer a fondo su funcionamiento y el de las pseudovecindades PHASES y HV para las cuales tiene que crear ciclos especiales de ejecución.

Por otra parte, en los experimentos realizados con CAMEX que no involucren a la vecindad de Margolus, se puede prescindir por completo de las pseudovecindades HV y PHASES, puesto que el trabajo que con ellas se realiza puede ser sustituido utilizando los recursos de los programas en REC, como se verá en la Sección 4.6.

4.3.2.1 Ejemplo ilustrativo: la función mooren

La función mooren es una de las funciones más importantes dentro de CAMEX debido a que permite el manejo en CAM de la vecindad de Moore. Para empezar, hay que indicar que esta función se halla contenida en el archivo CAMTA.C, y su código es el siguiente:

```
/* set parameters for Moore neighborhood */
mooren(l) int l; {
  wait();
  pokeb(BASE,TAA,MOORE); pokeb(BASE,TBA,MOORE);
  switch (l) {
```

```
    case 0:  pokeb(BASE,AAR,0x00); break;        /* planes 0,   1   */
    case 1:  pokeb(BASE,AAR,0x40); break;        /* planes 0a,  1a  */
    case 2:  pokeb(BASE,AAR,0x80); break;        /* planes 2a,  3a  */
    case 3:  pokeb(BASE,AAR,0xC0); break;        /* planes 02a, 13a */
    default: pokeb(BASE,AAR,0x00); break;}       /* planes 0,   1   */
}
```

La función en sí maneja un argumento 1 del cual nos ocuparemos más adelante, por el momento es conveniente analizar las primeras instrucciones que conforman el cuerpo de la función. La primera instrucción es simplemente una llamada a la función `wait` que suspende momentánemamente la actividad de la tarjeta CAM, y en el segundo renglón nos encontramos ya con las llamadas a `pokeb` que indican a la tarjeta que se va a trabajar con la vecindad de Moore; la primera llamada a `pokeb` instala la vecindad de Moore en CAM-A ya que escribe en el registro TAA (ver Figura 4.17) y la segunda hace lo propio en CAM-B al escribir en el registro TBA. El valor que se escribe en ambos registros es `MOORE`, que se halla definido en el archivo `CAMPC.H` como el número hexadecimal 16h[8]. Cambiando el valor anterior a binario obtenemos el número 00010110, que activa los bits TAS1, TAS2 y TAS4 del registro TAA (ver nuevamente la Fig. 4.17), y los bits TBS1, TBS2 y TBS4 del registro TBA. Con la activación de dichos bits se define la vecindad de Moore de la manera en que se definen vecindades en la CAM-PC (ver Secciones 2.2.2 y 4.1.2.3), es decir, los cables conectados a las entradas de la tabla de consulta (Sección 2.2.1) son conectados por el otro extremo a las celdas que conforman la vecindad de Moore. El argumento `MOORE` incluye entonces —en CAM-A— a las células noroeste, noreste, suroeste, sureste, norte, sur, oeste, y este del Plano 0, así como a los centros de CAM-B, que son el centro del Plano 2 y del Plano 3 (en otras palabras, es la vecindad de Moore junto con la pseudovecindad de `CENTERS` —por ello el 1 que activa al bit TAS4). Reconoceremos a las células antes mencionadas por NW0, NE0, SW0, SE0, N0, S0, W0, E0, C2 y C3. Para definir la vecindad de Moore en CAM-B cambiaríamos, evidentemente, el Plano 0 por el Plano 2, y los centros de CAM-A por los de CAM-B. Por cierto, es necesario mencionar que la selección anterior de células es sólo una parte de la vecindad de Moore, la cual

[8]En algunos casos se representarán los números en base 16 con una letra "h" al final. No se sigue una notación similar para los números binarios ya que una cadena relativamente larga de 1's y 0's no es fácil de confundir con un número decimal. De cualquier forma cuando aparece un número binario en el texto, explícitamente se menciona su base.

hay que completar con la asignación del valor 0 al bit LDTBL del registro CCR, ya que esto permite que la CAM disponga de los centros de cada plano (C0, C1, C2 y C3).

7	6	5	4	3	2	1	0
0	0	0	1	0	1	1	0
TAC1	TAM	TAS5	TAS4	TAS3	TAS2	TAS1	TAS0

Figura 4.17: Configuración de bits en el registro TAA que instala la vecindad de Moore en CAM-A.

Prácticamente lo anterior permite llevar a cabo el trabajo con la vecindad de Moore, y después se ejecuta la sentencia `switch` —en la función `mooren`—, que contempla 4 casos distintos en los que escribe algún valor en el registro AAR. Los valores que se escriben en cada caso se muestran en la Figura 4.18.

Cuando el Procesador de la CAM está en modo de procesamiento (Sección 2.1.2) lee información del Buffer de Planos, transforma la información, y vuelve a escribir los datos actualizados en el Buffer de Planos. Las tablas de consulta constituyen la parte del procesador de la CAM que está encargada de evaluar los vecindarios para determinar un nuevo estado y, como se vio en la Sección 2.2.1, tienen cuatro bits de salida; dos de los cuales identificamos como tablas primarias (etiquetados como "Planos"), y los otros dos como salidas auxiliares. Es precisamente de esos bits de salida de donde se obtienen los nuevos estados que se reescribirán en el Buffer de Planos, pero no toda la información (los cuatro bits de salida) es utilizada, y depende de los bits PCLA y PCLB del registro AAR la decisión de cuáles salidas de la tabla serán consideradas para reescribir el Buffer de Planos. Las configuraciones del registro AAR que se muestran en la Figura 4.18 se combinan con las configuraciones de los registros TAA y TBA, produciendo resultados varios que se pueden describir en forma extensa de la siguiente manera:

caso 0 la entrada del Buffer de Planos está dada por la salida de las tablas primarias tanto en CAM-A como en CAM-B

caso 1 la entrada de los planos 0 y 1 del Buffer de Planos la dan las tablas auxiliares 0 y 1. En lo que toca a los planos 2 y 3 la entrada la siguen proporcionando las tablas primarias 2 y 3.

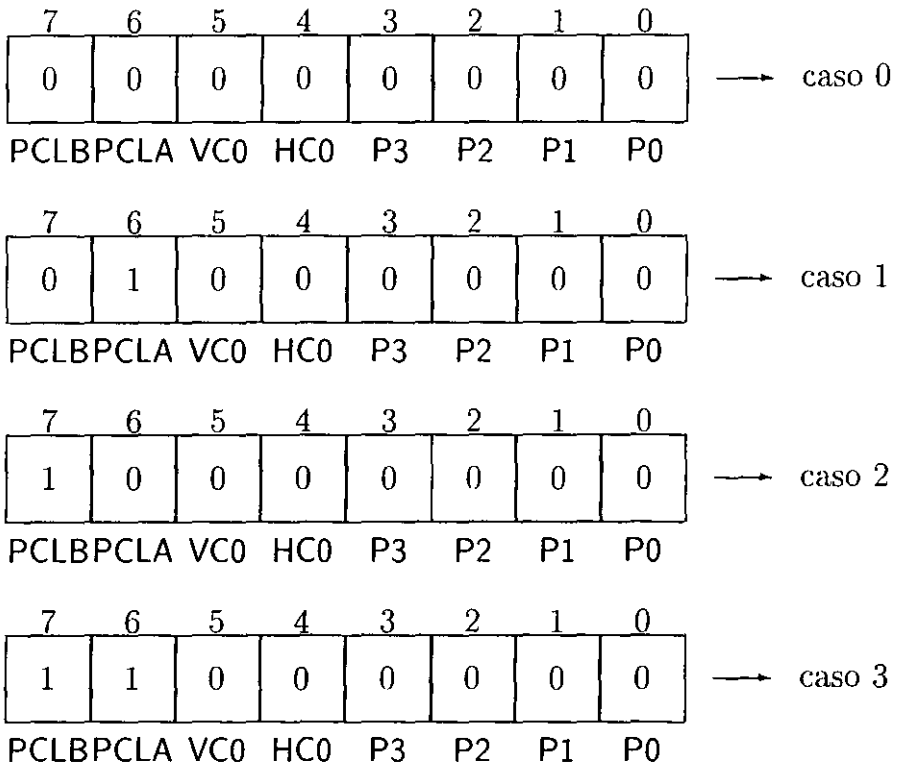


Figura 4.18: Configuraciones de bits en el registro AAR para los diferentes casos enunciados en la función `mooren`.

caso 2 la entrada de los planos 2 y 3 del Buffer de Planos la dan las tablas auxiliares 2 y 3. En lo que toca a los planos 0 y 1 la entrada la siguen proporcionando las tablas primarias 0 y 1.

caso 3 la entrada del Buffer de Planos está dada por la salida de las tablas auxiliares tanto en CAM-A como en CAM-B

4.3.3 Tablas de consulta

Lo último que falta por explicar es el manejo de las tablas de consulta: cómo determina y crea el usuario su contenido, y cómo estos datos creados por el usuario son introducidos físicamente en las tablas que se hallan dentro del Procesador CAM. Llamaremos a la primera operación 'creación' de las tablas, y a la segunda 'llenado' de las tablas.

4.3.3.1 Creación de las tablas

Este proceso puede ser considerado como la parte 'lógica' del tratamiento de las tablas de consulta, pues constituye una abstracción de las reglas del autómata celularque permiten determinar la evolución de cada configuración específica de los vecindarios. Aquí no se trabaja con ninguno de los registros de la CAM-PC, y sólo es necesario tener en cuenta qué tipo de vecindad vamos a usar y cuál es la nomenclatura de cada vecino, todo esto con el fin de mantener la consistencia a través del programa entero.

Además de no trabajar con registros, la creación de las tablas sólo debe contemplar las asignaciones mayores, ya que las pseudovecindades —como se intuye de su nombre— no contienen verdaderos vecinos. De hecho, es hasta la etapa de llenado de las tablas que se contempla la manera en que nos podemos auxiliar de los pseudovecinos.

Supongamos que queremos establecer en CAM-A un autómata celularque tenga dos estados, que use la vecindad de Moore y que la regla que determina su dinámica de evolución sea una totalística en la cual, por decir, si la suma de los estados de los vecinos es igual a 5 la célula evaluada (es decir, la célula central) estará en el estado 1 en la siguiente generación, y en el estado 0 en cualquier otro caso. Para crear una tabla que sea compatible con la CAM-PC debemos fijarnos primero que en ella la vecindad de Moore consta de 10 vecinos, y aunque no vayamos a ocupar en este experimento la célula central del plano 1, nuestra tabla debe contemplar las 2^{10}=1024 diferentes

configuraciones correspondientes a la vecindad de Moore. Para lograr esto, en el código de CAMEX definiremos arreglos unidimensionales de longitud igual a 1024, donde cada elemento corresponde a una disposición específica de la vecindad, y donde el valor contenido por cada elemento es el que produce la regla del autómata celularal ser aplicada en dicha disposición.

En base a lo anterior, la creación de la tabla correspondiente al autómata celularde nuestro ejemplo se hará utilizando un arreglo `t` que conste de 1024 elementos, y diez ciclos `for` que vayan de 0 a `KK`, la cual es igual a 2 y está definida en el archivo `CAMEX.H`. De este modo, el código para crear la tabla es el siguiente:

```
for (a0=0; a0<KK; a0++)
for (a1=0; a1<KK; a1++)
for (a2=0; a2<KK; a2++)
for (a3=0; a3<KK; a3++)
for (a4=0; a4<KK; a4++)
for (a5=0; a5<KK; a5++)
for (a6=0; a6<KK; a6++)
for (a7=0; a7<KK; a7++)
for (p0=0; p0<KK; p0++)
for (p1=0; p1<KK; p1++){

      s=a0+a2+a3+a4+a5+a6+a7+p0+p1;
      if (s==5)(s=1);else (s=0);
      t[tenq(a0,a1,a2,a3,a4,a5,a6,a7,p0,p1)]=s;
}
```

donde `tenq` toma los valores de los índices de los diez ciclos para formar un entero entre 0 y 1023, que corresponderá a cada una de las configuraciones en la tabla de consulta. Nótese que las variables que sirven de índices para los ciclos corresponden a los identificadores de los vecinos en CAMEX, tal y como se estableció en la Tabla 4.11.

4.3.3.2 Llenado de tablas: la función `camtbl`

Desde el momento en que la operación de llenado de tablas implica la escritura en registros, más el cuidado que hay que poner en lo tocante a los pseudovecinos, la situación que manejamos se complica, y tal vez sea necesario leer de nuevo la parte de la Sección 4.1.2.4, referente a TDAT.

Antes de comenzar nuestra explicación sobre el llenado de las tablas hay que tener presente que esto debe hacerse durante el modo ocioso, y que una vez realizado es necesario regresar al modo de despliegue. Para ello CAMEX cuenta con funciones que permiten entrar o salir de los diferentes modos de operación, y que deben usarse de la siguiente manera:

```
wait();
notv();
tloa();

AQUI SE DEBEN HACER LAS LLAMADAS A LA FUNCION QUE
LLENA LAS TABLAS

gotv();
```

donde

- `wait` provoca una pausa en la operación de la tarjeta;
- `notv` suspende el dibujado en pantalla mientras se cargan las tablas;
- `tloa` prepara las líneas de dirección de entrada para que se puedan cargar las tablas; y
- `gotv` regresa la tarjeta al modo de despliegue.

En CAMEX el llenado de las tablas se lleva a cabo mediante la función `camtbl`, que se halla en el archivo `CAMSU.C` y cuyo código es el siguiente:

```
/* ------------------------------------------ */
/* combine evolution tables into a CAM table row */
/* each byte packs two nibbles CAMB|CAMA      */
/* each nibble packs two pairs plane|aux      */
/* the pairs are bit planes A=0|1, B=2|3;     */
/* the aux's carry additional data            */
/* ------------------------------------------ */
/* a row varies address bits a0-a7            */
/* phase bits p0-p3 complete the full 12-bits */
/* the kind of neighborhood is governed by ldtbl */
/* and tas(tbs) 0-6; tac and tbc vary planes &c  */
/* ------------------------------------------ */
```

```
camtbl(p2,p3,t0,t1,t2,t3,t4,t5,t6,t7)
char  *t0, *t1, *t2, *t3, *t4, *t5, *t6, *t7;
int  p2, p3; {
int  s, t, u;
int  a0, a1, a2, a3, a4, a5, a6 ,a7;
int  p0, p1;

  for (p0=0; p0<KK; p0++)
  for (p1=0; p1<KK; p1++) {
    pokeb(BASE,AAR,(char)nibq(p0,p1,p2,p3));
  for (a0=0; a0<KK; a0++)
  for (a1=0; a1<KK; a1++)
  for (a2=0; a2<KK; a2++)
  for (a3=0; a3<KK; a3++)
  for (a4=0; a4<KK; a4++)
  for (a5=0; a5<KK; a5++)
  for (a6=0; a6<KK; a6++)
  for (a7=0; a7<KK; a7++)
    {
    s=bytq(a0,a1,a2,a3,a4,a5,a6,a7);
    u=tenq(a0,a1,a2,a3,a4,a5,a6,a7,p0,p1);
    t=bytq(t0[u],t1[u],t2[u],t3[u],t4[u],t5[u],t6[u],t7[u]);
    pokeb(BASE,TDAT+s,(char)t);
    } }
}
```

El código de esta función es más complicado que el de las demás funciones que se han visto hasta el momento. Esta complejidad tiene su raíz en el hecho de que maneja las líneas de dirección de entrada de las tablas de consulta, por lo que requiere del manejo de argumentos para dos registros: TDAT y AAR. Como sabemos, el primero maneja las primeras 8 líneas de dirección de entrada, en tanto que el segundo activa las últimas 4.

Se puede ver que los argumentos de esta función son 10, siendo los primeros 2 simples números enteros que se refieren a los pseudovecinos, y los siguientes 8 son los arreglos unidimensionales que contienen las reglas de evolución creadas en la sección anterior, y están asociados a cada una de las 8 tablas de consulta contenidas en la CAM-PC. Hay que señalar que la correspondencia entre arreglos y tablas de consulta está dada como se muestra en la siguiente tabla:

Arreglo	Tabla de consulta
t0	Primaria 0
t1	Auxiliar 0
t2	Primaria 1
t3	Auxiliar 1
t4	Primaria 2
t5	Auxiliar 2
t6	Primaria 3
t7	Auxiliar 3

Tabla 4.13: Correspondencia entre los argumentos de `camtbl` y las tablas de consulta

La función `camtbl` contiene 10 ciclos `for` anidados cuyos valores cambian entre 0 y 1, proporcionándonos 1024 combinaciones diferentes al juntar los 10 índices. Cada índice está asociado a cada uno de los componentes de las vecindades (Sección 4.3.2, Tabla 4.11).

Entre el segundo y el tercer ciclo se encuentra la instrucción

```
pokeb(BASE,AAR,(char)nibq(p0,p1,p2,p3));
```

que al ejecutarse activa las líneas de dirección de entrada 8, 9, 10 y 11, al colocar en el registro AAR el valor de la función `nibq`, que toma cuatro dígitos con valores de 0 y 1 que forman un número binario de 4 cifras (un *nibble*, medio byte) y los transforma a un número en base decimal.

Ahora centremos nuestra atención en las 4 instrucciones que se hallan contenidas en los 10 ciclos:

```
s=bytq(a0,a1,a2,a3,a4,a5,a6,a7);
u=tenq(a0,a1,a2,a3,a4,a5,a6,a7,p0,p1);
t=bytq(t0[u],t1[u],t2[u],t3[u],t4[u],t5[u],t6[u],t7[u]);
pokeb(BASE,TDAT+s,(char)t);
```

En la primera, la función `bytq` toma los valores de los índices de los 8 ciclos más internos y forma con ellos un número decimal entero que es asignado a `s`, en tanto que en la segunda `tenq` hace lo mismo pero para los 10 índices,

asignando el valor decimal a la variable u.[9] Las funciones `bytq` y `tenq` lo que hacen, a grandes rasgos, es tomar una serie o cadena de valores 0 y 1 (ocho la primera y diez la segunda), los cuales son tomados como si formaran un número binario, y dentro de cada función las cadenas formadas por ceros y unos son multiplicadas por potencias de dos, de donde se obtienen los valores decimales que regresan ambas funciones.

Los diversos valores que va tomando `u` van sirviendo como índices de los 8 arreglos que contienen las reglas. Los elementos de cada arreglo son ceros y unos, y se juntan para ser convertidos en un número decimal que es igualado a la variable `t`. Esta última variable es la que se escribe directamente en el registro TDAT —i.e. es el valor que corresponde a alguno de los 4096 registros que conforman las tablas de consulta— mediante la instrucción

```
pokeb(BASE,TDAT+s,(char)t);
```

Esta última instrucción merece ser examinada con bastante cuidado, ya que aquí se puede ver la razón por la cual TDAT está constituido por 256 posiciones de memoria (*bytes*). El segundo argumento de la función `pokeb` es una constante (definida en el archivo `CAMPC.H`) que representa a la primera posición del registro TDAT en la tarjeta CAM-PC. Su valor es `0x400`, y entonces al irle sumando el valor de `s` nos iremos colocando en cada una de las posiciones que ocupa. Cada una de estas 256 posiciones representa a alguna de las configuraciones que se obtienen al mezclar los valores de las primeras 8 líneas de dirección de entrada, y de hecho, es al irse incrementando TDAT cuando éstas se van activando. Entonces, la activación de estas líneas de dirección de entrada mezcladas con la activación de las líneas de dirección manejadas con AAR nos dan el total de los 4096 registros que componen las tablas de consulta.

Por último, es necesario mencionar que `camtbl` debe ser invocada 4 veces, para abarcar las 4 posibles combinaciones de los bits que forman las pseudovecindades, esto para tener las tablas completas aunque no necesitemos de los pseudovecinos. No es recomendable llenar por completo las tablas en una sola función que incorpore 12 ciclos anidados en lugar de 10, ya que estos dos ciclos que se agregarían representan a los pseudovecinos y su uso implica que el usuario esté al tanto de su manejo. Por ejemplo, la pseudovecindad `PHASES`

[9] Las funciones `bytq` y `tenq` forman parte de CAMEX, y basta con llamarlas simplemente desde el programa. De hecho, todas las funciones mencionadas en esta tesis están incluidas en CAMEX.

permite trabajar con reglas distintas en pasos distintos de la evolución del autómata, y entonces supongamos que queremos trabajar con la regla `r1` en el plano 0 en los pasos que sean pares, y con la regla `r2` en los pasos impares; las demás tablas contendrán cualquier otra regla (por ejemplo, una que asigne solamente ceros en los demás planos, para no obstruir la visibilidad del plano 0), a la que denotaremos como `or`. Bastará con utilizar un sólo pseudovecino cuyo valor igual a 0 indicará un paso par, en tanto que su valor igual a 1 indicará un paso impar. De este modo la función `camtbl` es llamada 4 veces como sigue:

```
camtbl(0,0, r1,or,or,or, or,or,or,or);
camtbl(0,1, r2,or,or,or, or,or,or,or);
camtbl(1,0, r1,or,or,or, or,or,or,or);
camtbl(1,1, r2,or,or,or, or,or,or,or);
```

En los casos en que no se utilice la información proporcionada por los pseudovecinos resulta innecesario hacer una distinción entre los argumentos de `camtbl`, pero aún así no se recomienda crear una función que contenga 12 ciclos en lugar de los 10 que maneja `camtbl`. Para estos casos fue creada la función `camall`, que también está dentro de `CAMSU.C`, y cuyo código se muestra a continuación:

```
/* install four camtbl's, each with same data */
camall(t0,t1,t2,t3,t4,t5,t6,t7)
char  *t0, *t1, *t2, *t3, *t4, *t5, *t6, *t7; {
  wait();
  notv();
  tloa();
  camtbl(0,0, t0,t1,t2,t3, t4,t5,t6,t7);
  camtbl(0,1, t0,t1,t2,t3, t4,t5,t6,t7);
  camtbl(1,0, t0,t1,t2,t3, t4,t5,t6,t7);
  camtbl(1,1, t0,t1,t2,t3, t4,t5,t6,t7);
  gotv();
}
```

Como se puede ver, esta función llama cuatro veces a `camtbl`, asignando las mismas 8 tablas a las 4 combinaciones distintas de los bits que representan a los pseudovecinos.

4.4 El Juego de la Vida en CAMEX

Las anteriores secciones han dado información sobre la implementación en CAMEX de los diferentes elementos que conforman un autómata celular, y ahora veremos como está constituido un operador REC que instala todo lo necesario para trabajar con el Juego de la Vida, excepto la configuración inicial, debido a que es más conveniente tener otros operadores que instalen diferentes configuraciones iniciales, los cuales se pueden mezclar con el operador de la regla (ver Sección 1.2 para recordar las reglas de este autómata). Del mismo modo, los operadores que instalan configuraciones iniciales pueden mezclarse con operadores de otras reglas de autómatas celulares.

Empezaremos mencionando que existe una tabla[10] contenida en el archivo `CAMTBL.H`, que contiene todos los operadores y predicados en REC y que consta de tres columnas: en la primera se indica el tipo de función que se va a compilar (predicado, operador, operador con argumento, etc.), en la segunda se indica el nombre de la función que se va a ejecutar, y en la tercera y última se coloca un comentario que indica el carácter al que se halla asociado el operador y en el que se recomienda escribir un breve enunciado que explique el proceso que se desarrolla al ejecutarlo. Así, en dicha tabla encontramos que el operador de Vida se describe de la siguiente manera:

```
r_oper1,     ropl1,     "l - Conway's Life plane 0     ",
```

donde

- `r_oper1` nos dice que la función se va a compilar como un operador simple (i.e. un operador sin argumento),
- `ropl1` es el nombre de la función que se va a ejecutar, i.e., la función que se declara y se ejecuta en C, y
- `"l - Conway's Life plane 0 "` indica que el operador a usar en nuestros programas en REC será `l`, que instala la regla del Juego de la Vida de Conway.

Las funciones que se ejecutan se hallan en el archivo de `CAM.C`, y así el cuerpo de `ropl1` es

```
ropl1() {conlif()};
```

[10]Esta tabla es un arreglo de estructuras.

Esta función simplemente llama a otra función, identificada con el nombre de `conlif`, y que se halla en el archivo `CAMSU.C`. Es esta función la que contiene todo lo necesario para instalar la regla del Juego de la Vida, y su forma es la siguiente:

```
/* Instala la tabla del Juego de la Vida */
/* de Conway en el plano 0, con eco o    */
/* trace (rastro) en el plano 1          */
conlif(); {char  r2[TLEN], zr[TLEN];
  ceros(zr);              /*  regla que transforma todo en cero */
  lifetab(r0);            /*  regla  del juego de la Vida */
  tracetab(r1);           /*  regla para generar trace (rastro) */
  echotab(r2);            /*  regla para generar eco */

  camall(r0,r0,r1,r2,zr,zr,zr,zr);

  /* Se llama a mooren para instalar la vecindad de Moore, */
  /* la que no sea usada deber aparecer comentada dentro  */
  /* del codigo.                                          */
  mooren(1);              /*  modalidad para usar eco */
  mooren(0);              /*  modalidad para usar trace (rastro) */
}
```

Las variables que contienen la función son arreglos de longitud `TLEN`=1024, y es en ellos en los que se almacenarán las reglas del autómata celularque mediante `camall` son cargadas en CAM-PC. Es necesario señalar que los arreglos `r0` y `r1` no están declarados dentro de la función, sino que están declarados al principio del archivo `CAMSU.C` como variables externas, puesto que casi todas las funciones que instalan reglas de autómatas necesitan de al menos 2 arreglos, así que es conveniente declararlos globalmente.

Cada arreglo es tomado como argumento de una función que coloca en él una regla que cumplirá una labor específica en alguno de los planos de bits, y en la siguiente sección se explica qué regla genera cada función.

4.4.1 Generación de la tabla con la regla de Vida

Todas las funciones que se utilizan para generar las reglas que se utilizan en `conlif` se hallan en el archivo `CAMTAM.C`, y de todas ellas examinaremos primero a `lifetab`, pues con esta se crea la regla del Juego de la Vida. El código completo de la función `lifetab` es el siguiente:

```
lifetab(t) char *t; {
int  a0, a1, a2, a3, a4, a5, a6 ,a7, p0, p1;
int  s;
  for (a0=0; a0<KK; a0++)
  for (a1=0; a1<KK; a1++)
  for (a2=0; a2<KK; a2++)
  for (a3=0; a3<KK; a3++)
  for (a4=0; a4<KK; a4++)
  for (a5=0; a5<KK; a5++)
  for (a6=0; a6<KK; a6++)
  for (a7=0; a7<KK; a7++)
  for (p0=0; p0<KK; p0++)
  for (p1=0; p1<KK; p1++)
    {

    s=a2+a3+a4+a5+a6+a7+p0+p1;
    if (a0==0) {if (s==3) s=1; else s=0;}
    if (a0==1) {if (s==2 || s==3) s=1; else s=0;}

    t[tenq(a0,a1,a2,a3,a4,a5,a6,a7,p0,p1)]=s;
    }
}
```

Los índices de cada ciclo `for` están plenamente identificados con los componentes de la vecindad de Moore (ver Tabla 4.11), y la regla del autómata celularestá expresada en las siguientes líneas:

```
s=a2+a3+a4+a5+a6+a7+p0+p1;
if (a0==0) {if (s==3) s=1; else s=0;}
if (a0==1) {if (s==2 || s==3) s=1; else s=0;}
```

4.4.1.1 Las funciones `echotab` y `tracetab`

En el segundo capítulo se habló sobre la posibilidad de complementar el Juego de la Vida con reglas para hacer *eco* y *rastreo*, las cuales nos brindan cierta información visual adicional sobre la evolución de algún autómata. Ambas reglas son asignadas a un segundo plano, y así, mientras el Juego de la Vida corre en el plano 0 o en el plano 2 (según se trate de CAM-A0 CAM-B), cualquiera de estas correrá en el plano 1 o en el plano 3.

En sí, la regla que hace *eco* muestra en el segundo plano de bits las células que estuvieron vivas en el primer plano en la generación inmediata anterior. La función `echotab` es la que genera el *eco* mediante las siguientes instrucciones:

```
/* Tabla para la regla de Eco */

echotab(t) char *t; {
int  a0, a1, a2, a3, a4, a5, a6 ,a7, p0, p1;
int  s;
  for (a0=0; a0<KK; a0++)
  for (a1=0; a1<KK; a1++)
  for (a2=0; a2<KK; a2++)
  for (a3=0; a3<KK; a3++)
  for (a4=0; a4<KK; a4++)
  for (a5=0; a5<KK; a5++)
  for (a6=0; a6<KK; a6++)
  for (a7=0; a7<KK; a7++)
  for (p0=0; p0<KK; p0++)
  for (p1=0; p1<KK; p1++)
    {
    s=a0;
    t[tenq(a0,a1,a2,a3,a4,a5,a6,a7,p0,p1)]=s;
    }
}
```

El *eco* consiste simplemente en colocar el valor de la célula central del primer plano antes de ser transformada en el segundo, y esto se logra con la instrucción

```
s=a0;
```

donde `a0` es la célula central del primer plano. Hay que notar que nunca se hace uso de `a1`, la célula central del segundo plano.

Por su parte, el *rastreo* o *tracing* cumple con una labor similar a la de *eco*, pero podríamos decir que tiene una *memoria* más larga que este último, pues lo que se busca es que alguna célula que estuvo viva en algún momento en el primer plano deje *huella* de su existencia en forma permanente en el segundo plano. El *rastreo* se lleva a cabo con la función que se muestra a continuación:

```
/* Tabla de la regla para dejar rastro  */

tracetab(t) char *t; {
int  a0, a1, a2, a3, a4, a5, a6 ,a7, p0, p1;
int  s;
  for (a0=0; a0<KK; a0++)
  for (a1=0; a1<KK; a1++)
  for (a2=0; a2<KK; a2++)
  for (a3=0; a3<KK; a3++)
  for (a4=0; a4<KK; a4++)
  for (a5=0; a5<KK; a5++)
  for (a6=0; a6<KK; a6++)
  for (a7=0; a7<KK; a7++)
  for (p0=0; p0<KK; p0++)
  for (p1=0; p1<KK; p1++)
    {
    s=a0|a1;
    t[tenq(a0,a1,a2,a3,a4,a5,a6,a7,p0,p1)]=s;
    }
}
```

donde la instrucción

```
s=a0|a1;
```

produce el *rastro* al ir añadiendo mediante la operación lógica OR las células centrales del primer y el segundo plano. Dado que las células de los planos son bits, se usa el operador |, que en lenguaje C realiza la operación OR en bits.

4.4.1.2 La función ceros

Esta función simplemente crea una tabla que hace evolucionar cualquier configuración del autómata celulara un estado nulo. En otras palabras, esta regla "mata" al plano en el que es aplicada, y su utilidad práctica es la de evitar que los planos que no son usados interfieran con la simulación actual, ya sea que afecten directamente la dinámica del autómata celularo que simplemente distorsionen la visualización de éste al existir células desplegadas en el monitor que correspondan a ellos. Visto de otra forma, esta función es una opción para limpiar planos usando una regla de evolución. El código de esta función es:

```
/* Tabla para dejar el plano en 0 */

ceros(t) char *t; {
int  a0, a1, a2, a3, a4, a5, a6 ,a7, p0, p1;
int  s;
  for (a0=0; a0<KK; a0++)
  for (a1=0; a1<KK; a1++)
  for (a2=0; a2<KK; a2++)
  for (a3=0; a3<KK; a3++)
  for (a4=0; a4<KK; a4++)
  for (a5=0; a5<KK; a5++)
  for (a6=0; a6<KK; a6++)
  for (a7=0; a7<KK; a7++)
  for (p0=0; p0<KK; p0++)
  for (p1=0; p1<KK; p1++)
    {
    t[tenq(a0,a1,a2,a3,a4,a5,a6,a7,p0,p1)]=0;
    }
}
```

4.4.2 Asignación de las reglas a los planos

Todas las reglas que se han visto deben se asignadas a algún plano en especial. En nuestro caso queremos que la regla del Juego de la Vida trabaje en el plano 0, por lo que las reglas para realizar *eco* o *rastreo* deben estar en el plano 1. Los planos 2 y 3 deben entonces permanecer estáticos para no interferir con la evolución de nuestro autómata. Luego, de acuerdo a lo visto en la sección 4.3.3.2 la llamada a la función `camall` desde `conlif` es:

```
camall(r0,r0,r1,r2,zr,zr,zr,zr);
```

La llamada es `camall` porque no necesitamos de ninguna distinción de pseudovecinos, y al cambiar de 0 a 1 el argumento de la función `mooren` tendremos, en el plano 1 la regla para *rastreo* o la regla para *eco*, respectivamente.

4.4.3 Un programa en REC que permite trabajar con el Juego de la Vida

En el Capítulo 3 se mencionó la manera en que trabaja el lenguaje REC, e incluso se vieron algunos de los operadores de este lenguaje que existen en CAMEX. En la sección anterior se estudió el operador que instala la regla del Juego de la Vida, y es asignado al carácter ASCII 1. Un programa en REC que permita trabajar con este autómata celularsería el siguiente:

```
(zf R 1 (gk:;) ;)
```

donde

- `z` es un operador que con el argumento `f` limpia todos los planos de la CAM;
- `R` es un operador que coloca una elipse llenada aleatoriamente con células en los estados 1 (viva) y 0 (muerta), en el centro del Plano 0;
- `1` es el operador que instala la regla para el autómata celulardel Juego de la Vida; y
- `(gk:;)` es un ciclo *while* en el que se repite el operador `g` mientras que no se cumpla con la condición del predicado `k`. El operador `g` da un paso de la evolución en tanto que `k` monitorea la actividad del teclado y termina con el ciclo cuando alguna tecla es presionada. Al presionar alguna tecla, `k` (que tiene el valor de verdadero al iniciar el programa) toma el valor de falso y provoca que el flujo del programa salte al siguiente segmento del código, que en este caso es el fin del programa.

Una versión diferente de un experimento en CAMEX con la regla del Juego de la Vida es:

```
(zf n1 1 (gk:;) ;)
```

donde el único cambio está en la configuración inicial. En este caso la configuración inicial se instala con el operador `n`, que coloca células en los estados 1 y 0 de forma aleatoria en los planos. Este operador toma el argumento 1, lo cual quiere decir que instala las células en el Plano 0 (ver Tabla 4.10).

En las Figuras 4.19 y 4.20 se pueden ver evoluciones del Juego de la Vida, la primera muestra rastro y partió de una elipse como configuración inicial. La segunda muestra eco y tuvo como configuración inicial a varias células en los estados 1 y 0 esparcidas de manera aleatoria por todo el plano.

Figura 4.19: Imagen del Juego de la Vida con *rastro*

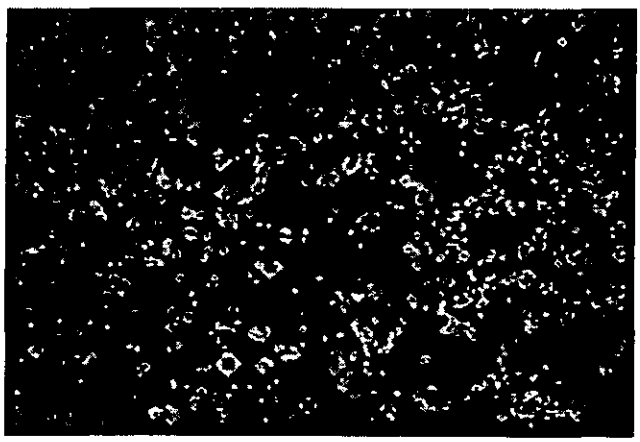


Figura 4.20: Imagen del Juego de la Vida con *eco*

4.5 Inserción de un nuevo operador de REC

La inserción de nuevos operadores en CAMEX, aunque tediosa, es bastante sencilla y se puede llevar a cabo siguiendo los pasos que a continuación enunciamos:

1. Abrir el archivo `CAMTBL.H` y encontrar algún carácter ASCII libre en la tabla de operadores y predicados de REC. Los caracteres libres tienen la siguiente forma en la tabla:

   ```
   r_noopc,      FALSE,      "N                                    ",
   ```

   (este ejemplo se refiere concretamente al caso de la letra 'N').

2. Una vez escogido el carácter ASCII que contendrá nuestro nuevo operador debemos darlo de alta en la tabla, recordando que en la primera columna se debe indicar el tipo de función de REC que se compilará (operador, predicado, operador con 1 argumento, etc.), en la segunda deberá ir el nombre de la función que se ejecuta (i.e. la función que se declara y se usa dentro del programa), y por último, en la tercera columna se debe añadir un comentario junto al carácter ASCII. Como ya se mencionó, a los operadores van asociadas funciones escritas en C, que en general realiza cualquier tarea que se desee; como la instalación de una configuración, una vecindad, o una regla de transición. En este apartado ejemplificamos con la instalación de la regla de un autómata celular.

3. En el mismo archivo `CAMTBL.H` se debe declarar la función ejecutable como de tipo entero, debajo del comentario

   ```
   /* ------------------------------------------------ */
   /* Declarations of user-provided execution subroutines */
   /* ------------------------------------------------ */
   ```

4. Abrir `CAM.C`, el cual tiene en su parte final una sección dedicada a los componentes de REC bajo el siguiente comentario:

   ```
   /* ----------------------------------------*/
   /* definition of REC operators and predicates */
   /* ----------------------------------------*/
   ```

En esta sección se definirá (como lo indica el comentario) nuestro nuevo operador, esto es, aquí se encontrará el cuerpo de la función que declaramos en el punto anterior. Para mantener el código uniforme y legible se recomienda que los operadores estén definidos en base a otra función que deberá declararse y definirse en otro archivo del programa.

5. La función en base a la cual está definida la función ejecutable REC debe crearse en otro archivo, por ejemplo, puede incluirse en `CAMSU.C`. En esta función es en donde se da prácticamente el código que describe a nuestro autómata, pues en ella se deben incluir las llamadas a las funciones que crean las tablas de consulta y a aquellas que permiten llevar a cabo la visualización en pantalla de la evolución del autómata.

6. Por último, el usuario debe crear la tabla de consulta que dará la información para que se realicen las transformaciones en el autómata. Para ello sólo hay que ajustar a nuestro autómata celularparticular la fórmula que es asignada a s en la siguiente estructura:

```
/* ------------------------------------------*/
/*      --- Generacion de una tabla ---      */
/*  ESTE EJEMPLO CREA LA TABLA PARA EL       */
/*  AUTOMATA VIDA DE CONWAY.  ESTE COMENTARIO */
/*  DEBE REEMPLAZARSE A FIN DE DOCUMENTAR    */
/*  NUESTRO AUTOMATA.                        */
/* ------------------------------------------*/

nombre_de_tabla(t) char *t; {
int  a0, a1, a2, a3, a4, a5, a6 ,a7, p0, p1;
int  s;
  for (a0=0; a0<KK; a0++)
  for (a1=0; a1<KK; a1++)
  for (a2=0; a2<KK; a2++)
  for (a3=0; a3<KK; a3++)
  for (a4=0; a4<KK; a4++)
  for (a5=0; a5<KK; a5++)
  for (a6=0; a6<KK; a6++)
  for (a7=0; a7<KK; a7++)
  for (p0=0; p0<KK; p0++)
  for (p1=0; p1<KK; p1++)
    {
```

```
        s=a2+a3+a4+a5+a6+a7+p0+p1;
        if (a0==0) {if (s==3) s=1; else s=0;}
        if (a0==1) {if (s==2 || s==3) s=1; else s=0;}

        t[tenq(a0,a1,a2,a3,a4,a5,a6,a7,p0,p1)]=s;
        }
}
```

Esta tabla es conveniente incluirla en algún otro archivo, como podría ser `CAMTAM.C`, que es donde se encuentran declaradas las tablas de evolución para autómatas que funcionan con vecindad de Moore.

4.5.1 Ejemplo: Inserción del operador Paridad

La regla de Paridad fue sugerida por Edward Fredkin del MIT en los primeros años del desarrollo de los autómatas celulares, especificando que una célula "sigue la *paridad* de su vecindario", es decir, estará viva o muerta dependiendo de si su vecindario actual contiene un número par o impar de células vivas (Sección 1.7.5). Para la implementación de nuestra regla trabajaremos con la vecindad de Moore, pero usando sólo los elementos centro (`a0`), norte (`p1`), sur (`p0`), este (`a6`) y oeste (`a7`). La regla de Paridad puede traducirse como el OR exclusivo de los cinco elementos mencionados.

La razón por la que usamos la vecindad de Moore y no la de von Neumann (que se ajusta exactamente al modelo) es porque a fin de cuentas la primera incluye a la segunda (de hecho incluye a todas las vecindades de radio 1 en dos dimensiones), y se presta para llevar mejoras al autómata celular mediante la inclusión de reglas como la de rastreo, y además nos brinda la posibilidad de implementar en la misma función de CAMEX una regla de Paridad que tome en cuenta a los 9 elementos de la vecindad, lo cual nos permitiría tener dos autómatas diferentes en el mismo operador, pudiendo ocupar el que queramos mediante el uso de un argumento.

Los pasos que hay que seguir para insertar el operador de REC que permita trabajar con la regla de Paridad en el plano 0 son los siguientes:

1. En el archivo `CAMTBL.H`, buscar un carácter ASCII que esté vacío, por ejemplo, el carácter `s`:

```
r_noopc,      FALSE,      "s-                              ",
```

2. Sustituimos los elementos anteriores por las siguientes indicaciones:

```
r_code,     ropar,      "s - Regla de Paridad          ",
```

3. Declarar en `CAMTBL.H` a `ropar()` como de tipo entero:

```
/* ------------------------------------------------- */
/* Declarations of user-provided execution subroutines */
/* ------------------------------------------------- */

int ropar();
```

4. Dar de alta a `ropar()` en el archivo `CAM.C`, en la sección final de éste, definiéndola en base a otra función, a la que llamaremos `parity`:

```
ropar() {parity();}
```

5. Crear a la función `parity` en `CAMSU.C`:

```
/* Instala la regla de Paridad en el plano 0 */
parity(){char zr[TLEN];
ceros(zr);
partab(r0);          /* Tabla de busqueda para Paridad */
camall(r0,zr,zr,zr,zr,zr,zr,zr);
mooren(0);
}
```

6. Crear la tabla de consulta que describa la regla de Paridad en el archivo `CAMTA.C`, con el nombre de `partab()`:

```
/* Tabla de busqueda para la regla de Paridad */

partab(t) char *t; {
int  a0, a1, a2, a3, a4, a5, a6 ,a7, p0, p1;
int  s;
  for (a0=0; a0<KK; a0++)
  for (a1=0; a1<KK; a1++)
  for (a2=0; a2<KK; a2++)
  for (a3=0; a3<KK; a3++)
```

```
    for (a4=0; a4<KK; a4++)
    for (a5=0; a5<KK; a5++)
    for (a6=0; a6<KK; a6++)
    for (a7=0; a7<KK; a7++)
    for (p0=0; p0<KK; p0++)
    for (p1=0; p1<KK; p1++)
      {

      s=(a0^a6^a7^p0^p1);  /* Esta instruccion es Paridad */

      t[tenq(a0,a1,a2,a3,a4,a5,a6,a7,p0,p1)]=s;
      }
}
```

7. Compilar nuevamente el proyecto CAMEX para crear un nuevo programa ejecutable que contenga a nuestro nuevo operador **s**, que instala la regla de Paridad (ver el Apéndice A.1).

Figura 4.21: Imagen de la regla de Paridad de Fredkin

4.6 Uso de las pseudovecindades en CAMEX

Las pseudovecindades presentan bastantes ventajas, aunque en ocasiones requieren de esfuerzos extra de programación, los cuales pueden ser evitados aprovechando algunas características propias de CAMEX. En concreto, las

pseudovecindades que pueden ser emuladas por el programa son PHASES y HV.

Por ejemplo, PHASES es usada para trabajar con reglas compuestas y para ello utiliza dos pseudovecinos asociados a los bits P2 y P3 de AAR, los cuales deben ser activados y desactivados durante la simulación para lograr los efectos deseados. Supongamos que deseamos usar la pseudovecindad de PHASES para correr una regla compuesta en la que en los pasos pares se use la regla r1 y en los impares se use la regla r2. Pensando en que ya se han llenado correctamente las tablas de consulta, y que la selección de las vecindades es la adecuada, necesitaremos ahora de un ciclo que nos proporcione la información correspondiente a los pseudovecinos. El ciclo al que nos referimos debe tener la siguiente forma:

```
ciclo(){
    pokeb(BASE, AAR, 0x00);        /* P2=0, P3=0 */
    sstep();                       /* Funcion que da un paso de evolucion */
    pokeb(BASE, AAR, 0x04);        /* P2=1, P3=0 */
    sstep();
    pokeb(BASE, AAR, 0x08);        /* P2=0, P3=1 */
    sstep();
    pokeb(BASE, AAR, 0x0C);        /* P2=1, P3=1 */
    sstep();
}
```

Además, dicho ciclo debe ser asignado a un operador de REC, y ser ejecutado dentro de alguna estructura iterativa del tipo *while*, y aparte se debe tener cuidado en el llenado de las tablas de consulta, que en este caso quedarán como:

```
camtbl(0,0, r1,or,or,or, or,or,or,or);
camtbl(0,1, r2,or,or,or, or,or,or,or);
camtbl(1,0, r1,or,or,or, or,or,or,or);
camtbl(1,1, r2,or,or,or, or,or,or,or);
```

donde or son reglas indistintas para nuestro ejemplo.

Para evitar lo anterior y obtener resultados idénticos, se pueden utilizar dos métodos distintos en CAMEX:

1. Incluir en un ciclo iterativo operadores de REC que cambien las tablas de consulta primarias por las auxiliares. Dichos operadores simplemente necesitan estar asociados a las funciones que instalan las vecindades, pues estas ya tienen incorporadas opciones que permiten hacer los cambios mencionados. Por ejemplo, en REC existe el operador `m`, que está asociado a la función `mooren` (Sección 4.3.2.1), que al tomar como argumento el valor de `0` usa la tabla primaria del Plano 0, y al tomar el valor `1` usa la tabla auxiliar del Plano 0. Así, para alternar entre reglas podemos usar la función `camall` (que instala la mismas tablas para usar las combinaciones de pseudovecinos) y colocar la regla `r1` en la tabla primaria del Plano 0, y la regla `r2` en la tabla auxiliar del Plano 0. Después de esto, basta utilizar en un programa de REC el operador `m` con las formas `m0` y `m1`, alternándose ya sea en un ciclo *while* o mediante el uso del contador `!n!` (Sección 3.6.1).

2. Se pueden tener operadores de REC que carguen reglas diferentes, y entonces bastará con alternarlos en alguna de las formas mencionadas anteriormente. Esta opción presenta el inconveniente de que el monitor de video parpadeará cada que se haga el cambio de reglas, puesto que la carga de éstas implica entrar al modo ocioso y regresar al modo de despliegue (Sección 2.1.2).

En el caso de la vecindad `HV` la alternativa que presentamos no es precisamente una característica de CAMEX, sino que simplemente consiste en aplicar operaciones lógicas entre los bits de diferentes planos. Recordemos que el principal objetivo de `HV` es permitir que células en posiciones diferentes evolucionen de manera diferente de acuerdo a su posición horizontal y vertical en el plano. En otras palabras, `HV` permite que reglas de evolución distintas se apliquen en las diferentes posiciones de un plano que está cuadriculado como si fuera un tablero de ajedrez. En CAMEX existen las opciones `h` y `H` desde el entorno, y el operador `h` de REC, que permiten colocar una configuración cuadriculada (tablero de ajedrez, *checkerboard*) en uno o varios planos, la cual puede interactuar con alguna configuración existente en otro plano de acuerdo con alguna regla que hayamos instalado en las tablas de consulta. Se recomienda al lector correr la demostración de REC con el número 5, "`Life on a checkerboard`", que se halla en el menú que se despliega presionando `f2`.

Lo escrito en los párrafos anteriores no significa que en CAMEX no se necesitarán las pseudovecindades `PHASES` y `HV`; simplemente representan al-

ternativas a éstas cuando los experimentos son tan sencillos que tal vez no valgan el esfuerzo. Sin embargo la mayoría de los experimentos que utilizan la vecindad de Margolus necesitan forzosamente de estas pseudovecindades y de la creación de ciclos para su correcta ejecución.

4.6.1 Ejemplo ilustrativo sobre el uso de la psudovecindad PHASES y sus alternativas: el autómata celular *Borde/Hueco*

En el Capítulo 1 se explicó la forma en que trabaja el autómata celular *Borde/Hueco*, cuya característica principal es la de evolucionar mediante una regla compuesta, lo cual nos permite ver que el uso de la pseudovecindad de PHASES es ideal para este autómata celular. De este modo, mostraremos cómo se puede lograr la implantación de este autómata en CAMEX usando la pseudovecindad de PHASES, y después mostraremos la manera de implementar este mismo autómata celular usando las alternativas existentes en CAMEX para evitar el uso de pseudovecinos.

Utilizando la pseudovecindad preconstruida, podemos empezar estableciendo la vecindad de Moore (requerida por este autómata celular) para el uso de la pseudovecindad de PHASES. La función que llevará a cabo esta tarea será incluida en CAMTA.C, y la llamaremos moorph. Su funcionamiento es parecido al de la función mooren (ver Sección 4.3.2.1), y su código se muestra a continuación:

```
/* set parameters for Moore neighborhood with Phases */
moorph(l) int l; {
  wait();
  pokeb(BASE,TAA,MOCUR); pokeb(BASE,TBA,MOCUR);
  switch (l) {
    case 0:  pokeb(BASE,AAR,0x00); break;       /* planes 0,   1   */
    case 1:  pokeb(BASE,AAR,0x40); break;       /* planes 0a,  1a  */
    case 2:  pokeb(BASE,AAR,0x80); break;       /* planes 2a,  3a  */0
    case 3:  pokeb(BASE,AAR,0xC0); break;       /* planes 02a, 13a */
    default: pokeb(BASE,AAR,0x00); break;}      /* planes 0,   1   */
}
```

donde MOCUR es un valor constante que se halla en CAMPC.H, y que es el que permite el uso de los pseudovecinos de PHASES. La definición de MOCUR es la siguiente:

```
# define MOCUR  0x06 /* Moore neighborhood using phase rather than center */
```

A continuación debemos crear las reglas para crear el borde y para crear el hueco. Estas reglas se incluirán en `CAMTAM.C`, y de acuerdo a lo explicado en el Capítulo 1, tendrán la siguiente forma:

```
hollowtab(t) char *t; {
int  a0, a1, a2, a3, a4, a5, a6 ,a7, p0, p1;
int  s,s1;
  for (a0=0; a0<KK; a0++)
  for (a1=0; a1<KK; a1++)
  for (a2=0; a2<KK; a2++)
  for (a3=0; a3<KK; a3++)
  for (a4=0; a4<KK; a4++)
  for (a5=0; a5<KK; a5++)
  for (a6=0; a6<KK; a6++)
  for (a7=0; a7<KK; a7++)
  for (p0=0; p0<KK; p0++)
  for (p1=0; p1<KK; p1++)
    {
    s=(a2&a3&a4&a5&a6&a7&p0&p1);
    if(s==1)s1=0;
    if(s==0 && a0==1)s1=1;
    if(s==0 && a0==0)s1=0;
    t[tenq(a0,a1,a2,a3,a4,a5,a6,a7,p0,p1)]=s1;
    }
}

bordertab(t) char *t; {
int  a0, a1, a2, a3, a4, a5, a6 ,a7, p0, p1;
int  s;
  for (a0=0; a0<KK; a0++)
  for (a1=0; a1<KK; a1++)
  for (a2=0; a2<KK; a2++)
  for (a3=0; a3<KK; a3++)
  for (a4=0; a4<KK; a4++)
  for (a5=0; a5<KK; a5++)
  for (a6=0; a6<KK; a6++)
  for (a7=0; a7<KK; a7++)
  for (p0=0; p0<KK; p0++)
```

```
  for (p1=0; p1<KK; p1++)
    {
    s=(a0|a2|a3|a4|a5|a6|a7|p0|p1);
    t[tenq(a0,a1,a2,a3,a4,a5,a6,a7,p0,p1)]=s;
    }
}
```

Habiendo creado las tablas de evolución, y teniendo también creada la función que permite manejar la pseudovecindad, podemos crear ahora la función que permite la implantación de la regla en CAMEX, y que a fin de cuentas es la que será asociada a un operador de REC. Esta función se incluirá en `CAMSU.C`, y su código será el siguiente:

```
CAMSU.C
/* install the table for the HOLLOW/BORDER rule */
holbord() {
  char un[TLEN], zr[TLEN], r1[TLEN], r2[TLEN], r3[TLEN];

  ceros(zr,0,0,0,0);

  hollowtab(r2);
  bordertab(r3);

  wait();
  notv();
  tloa();
  camtbl(0,0, r3,r3,zr,zr, zr,zr,zr,zr);
  camtbl(0,1, r2,r2,zr,zr, zr,zr,zr,zr);
  camtbl(1,0, r3,r3,zr,zr, zr,zr,zr,zr);
  camtbl(1,1, r2,r2,zr,zr, zr,zr,zr,zr);
  gotv();

  moorph(0);

}
```

Como se mencionó, la función anterior será asignada a un operador de REC, y sólo nos faltaría elaborar la función que implantará el ciclo para aprovechar la información de los pseudovecinos. Dicho ciclo, como se mostró con anterioridad, se llamará `phcycle` tendrá la siguiente forma:

```
phcycle(){

pokeb(BASE,AAR,(char)0x00);
sstep();
pokeb(BASE,AAR,(char)0x04);
sstep();
pokeb(BASE,AAR,(char)0x08);
sstep();
pokeb(BASE,AAR,(char)0x0C);
sstep();
}
```

Este ciclo estará incluido en `CAM.C`[11], y también deberá asociarse a un operador de REC.

En lo referente a la asignación de operadores REC optamos por colocar la función que implanta la regla en el carácter ASCII `e`, y la función del ciclo en `f`. Así, un programa de REC que nos permita implantar un autómata celular con la regla de *Borde/Hueco* puede tener la siguiente forma:

```
(zf w1 e (fk:;) ;)
```

donde

- `z` es un operador que con el argumento `f` limpia todos los planos de la CAM;
- `w` es un operador que al tomar como argumento el valor de `1` coloca algunos puntos dispersos en el Plano 0;
- `e` es el operador que instala la regla para el autómata celular de Borde/Hueco; y
- `(fk:;)` es un ciclo *while* en el que se repite el operador `f` (ciclo que intercambia los valores de las fases) mientras que no se cumpla con la condición del predicado `k` (la presión de alguna tecla).

[11] En este punto deseamos recordar al lector que no es forzoso incluir las funciones en los archivos mencionados, sino que son meras sugerencias de nosotros para mantener cierta consistencia en la estructura del programa.

Si lo anterior se hubiera hecho utilizando las alternativas existentes en CAMEX (que fueron explicadas en esta sección), los programas de REC que nos permitirían trabajar con el autómata de *Borde/Hueco* podrían ser los siguientes[12]:

- Intercambiando tablas primarias y auxiliares[13].

```
(zf w1 e ((m0 g m1 g;)k:;) ;)
```

  donde

  - `z` es un operador que con el argumento `f` limpia todos los planos de la CAM;
  - `w` es un operador que al tomar como argumento el valor de 1 coloca algunos puntos dispersos en el Plano 0;
  - `e` es el operador que instala la regla para el autómata celular de Borde/Hueco; y
  - `((m0 g m1 g;)k:;)` es un ciclo *while* en el que se repite el conjunto de instrucciones `(m0 g m1 g;)`, estableciéndose con `m0` el uso de la tabla de consulta primaria (que debe ser *BORDE*), dándose un paso de evolución con `g`, para despus utilizar la tabla de consulta auxiliar (que será *HUECO*), y dar otro paso de evolución, y así hasta que se oprima alguna tecla.

- Intercambiando operadores de reglas.

```
(zf w1 ((B g H g;)k:;) ;)
```

  donde

  - `z` es un operador que con el argumento `f` limpia todos los planos de la CAM;

[12]Las tablas de consulta para las reglas son idénticas a las explicadas en los párrafos anteriores.

[13]Esto se hace usando la función `mooren` con argumentos que permitan el intercambio de tablas. Para mayor referencia ver la Sección 4.3.2.1.

- `w` es un operador que al tomar como argumento el valor de 1 coloca algunos puntos dispersos en el Plano 0; y
- `((B g H g;)k:;)` es un ciclo *while* en el que se repite el conjunto de instrucciones `(B g H g;)`, siendo `B` el operador REC que instala la regla de *BORDE*), dándose un paso de evolución con `g`, y después se usa el operador `H`, el cual instala la regla de *HUECO*, para después dar otro paso de evolución, y continuar así hasta que se oprima alguna tecla. Como se dijo, esta opción provoca "parpadeos" en la pantalla al ser puesto en ejecución el experimento.

4.7 Operadores con argumento

Los operadores con argumento representan una gran ventaja ya que permiten que una sola función pueda realizar diferentes tareas o realizar la misma tarea en distintos elementos. Un ejemplo de un operador que realiza la misma tarea en diferentes elementos es el que se encarga de limpiar los planos, el operador `z`, que al recibir como argumento algún valor hexadecimal entre `0` y `f` puede limpiar desde un sólo plano hasta todos juntos, incluyendo las distintas combinaciones que se puedan obtener de ellos.

Ahora bien, en lo que respecta a un operador que realice distintas tareas en base al argumento que reciba, lo único que hay que hacer es determinar la condición que evalúe el argumento y que decida a qué función llamar. Por ejemplo, se podría crear un operador que reciba un argumento y que en base a este decida si la regla de un autómata celular que se utilizará será la del Juego de la Vida en su forma más simple, o si ejecutará el Juego de la Vida con rastro.

Uno de los aspectos más prácticos del manejo de argumentos es que en un sólo operador se pueden tener varias reglas de autómatas celulares, lo que deja más caracteres ASCII libres para ser habilitados como otros operadores de REC. Como práctica sana, se recomienda que si un operador va a contener varias reglas, éstas sean afines, es decir, que en el mismo operador es recomendable tener la regla de Paridad para 5 y 9 vecinos, pero que sería demasiado confuso tener en el mismo operador a la regla del Juego de la Vida, a la regla de Paridad y a la de Borde/Hueco [14].

[14] Todas las reglas aquí mencionadas están explicadas en el Capítulo 1

Por ejemplo, en un solo operador podemos tener las reglas para los autómatas celulares de Cuadrados, Diamantes y Triángulos, y así, a un operador con argumento le podemos asignar la siguiente función —que de acuerdo con nuestras convenciones deberá estar en `CAMSU.C`—, cuyo argumento es el argumento del operador:

```
/* install the table for Squares, Diamonds & Triangles rule */
sdt(l) int l; {
  char un[TLEN], zr[TLEN];
  booltab(zr,0,0,0,0);
  booltab(un,1,1,1,1);
  sdttab(r0,l);                /* ...  rule for SDT */
  camall(r0,r0,zr,zr,zr,zr,zr,zr);
  mooren(1);
}
```

En este caso el argumento es pasado a otra función —incluida en `CAMTAM.C`— que es la que establece la regla a utilizar en base al argumento recibido:

```
/* Table for the Squares, Diamonds & Triangles rules */

sdttab(t,l) char *t; int l; {
int  a0, a1, a2, a3, a4, a5, a6 ,a7, p0, p1;
int  s;
  for (a0=0; a0<KK; a0++)
  for (a1=0; a1<KK; a1++)
  for (a2=0; a2<KK; a2++)
  for (a3=0; a3<KK; a3++)
  for (a4=0; a4<KK; a4++)
  for (a5=0; a5<KK; a5++)
  for (a6=0; a6<KK; a6++)
  for (a7=0; a7<KK; a7++)
  for (p0=0; p0<KK; p0++)
  for (p1=0; p1<KK; p1++)
    {
    if (l==0) (s=a0|a2|a3|a4|a5|a6|a7|p0|p1); /* SQUARES */
    if (l==1) (s=a0|a6|a7|p1|p0); /* DIAMONDS */
    if (l==2) (s=a0|a6|a7|p1); /* TRIANGLES */
    t[tenq(a0,a1,a2,a3,a4,a5,a6,a7,p0,p1)]=s;
    }
}
```

Se puede ver que cuando el argumento es igual a 0 se instalará en la CAM-PC la regla de Cuadrados (*Squares*), cuando sea igual a 1 se usará Diamantes (*Diamonds*), y en el caso de que el argumento sea 2 se instalará Triángulos (*Triangles*). Si el argumento no está contemplado en la función no se instalará ninguna regla en la tarjeta, la cual conservará la regla instalada por el último operador que se haya ejecutado.

4.8 Comentarios finales

CAMEX presenta bastantes ventajas y en este capítulo se abordaron algunos tópicos específicos que ejemplifican la manera en que el usuario puede aprovechar al máximo las capacidades de la tarjeta. No es tarea sencilla el programar los componentes de un autómata celular en CAMEX, pero con el conocimiento de los registros de la tarjeta, más conocimientos de lenguaje C y de la teoría básica de autómatas celulares, se puede conseguir un gran poder para la realización de experimentos. Por ejemplo, el *software* original muestra el trabajo con reglas compuestas (dos o más reglas aplicadas alternadamente en un espacio de células a través del tiempo) utilizando la pseudovecindad preconstruida de `PHASES`, mientras que un estudio detallado del funcionamiento de las vecindades y las reglas a nivel de *hardware* nos permitió visualizar la manera en que se puede llevar a cabo un experimento con reglas compuestas, mediante CAMEX, sin necesidad de recurrir a la pseudovecindad preconstruida antes mencionada, sino simplemente intercambiando las tablas de reglas (ver Sección 4.6). Aspectos como este y algunos otros hacen que CAMEX aparente estar programado de una manera más "elegante" que el *software* original de la CAM, aprovechando de una manera eficiente los recursos de la tarjeta en beneficio del usuario.

No obstante no podemos dejar de mencionar los aspectos en los que CAMEX es más deficiente que el programa original. Desventajas de CAMEX son la poca documentación (problema que este trabajo intenta ayudar a solucionar) y los *bugs*. CAMEX, como todos los programas grandes o/y ambiciosos, no está libre de los errores involuntarios (llamados *bugs* en inglés y *gazapos lógicos* en algunos textos en español) que se cometen al programar. Y aunque eso de ninguna manera merma las cualidades de este paquete, es conveniente hacer notar que este programa no está libre de éstos. Entre varios otros, los siguientes son algunos de los gazapos lógicos de CAMEX:

- El parámetro del checkerboard.
- El radio de la elipse se traba.
- No sirve el editor de planos.
- La opción de *Wireworld* en el menú asociado a `f1` no funciona.
- La opción para utilizar un ratón (*mouse*) no funciona.

Otro aspecto negativo es que es menos atractivo para el usuario que el *software* original, el cual es más amigable y las demostraciones que incluye de autómatas celulares son más variadas y visualmente más atractivas que las de CAMEX. Si un usuario desea experimentar un poco con autómatas celulares sin mucho trabajo es más aconsejable revisar el software original. Pero en el momento en que el usuario desee adentrarse más en la experimentación con autómatas celulares mediante la CAM-PC y a realizar sus propios experimentos, entonces tendrá que profundizar en el lenguaje Forth y en el conocimiento del funcionamiento de la tarjeta, y es en este momento en el que creemos que CAMEX toma ventaja, al ofrecer un código abierto y un lenguaje difundido para manipular directamente los recursos de la CAM-PC.

Por último, y a manera de moraleja, queremos hacer notar el hecho de que este capítulo resalta la importancia y el gran valor del *software* gratuito y de código abierto, aparte del gran beneficio que reporta al usuario el conocer la forma (en mayor o menor medida) en que trabajan sus programas y su *hardware*. Este tipo de programas es también de gran valor para los estudiantes de computación, ya que pueden obtenerlos sin cargo alguno y comprender y analizar cómo está elaborado un programa que controla un dispositivo *hardware* libremente y sacar provecho de esto, cosa que no se puede hacer con la gran mayoría de los paquetes de computadora, que pertenecen a la clase conocida como *software propietario*.

Conclusiones

Los autómatas celulares son modelos sumamente atractivos para investigadores de diversas áreas, pero que presentan el inconveniente de que una gran cantidad de cómputo es requerida para tener resultados significativos. Con el advenimiento de la CAM-PC, nuevos estudios pudieron realizarse en autómatas celulares en dos dimensiones, ya que esta máquina permite realizar experimentos con un alto grado de eficiencia a un costo muy bajo.

CAMEX ha mostrado tener considerables ventajas sobre el programa que se distribuye con la CAM-PC, lo cual lo hace una opción viable para usarlo como controlador de la tarjeta. Este programa fue desarrollado en México y permite explotar las capacidades de la CAM-PC de una forma total y flexible, pues es de distribución gratuita y de código abierto, lo cual da al experimentador la posibilidad de manipular la tarjeta a bajo nivel y al detalle deseado. Tomando en cuenta lo antes mencionado, podemos ver que el uso de ambas herramientas (CAM-PC y CAMEX) se presenta como una excelente oportunidad para realizar investigaciones y experimentos de alto nivel sobre autómatas celulares en países como el nuestro, que tienen como problema fundamental, entre otros tantos, la falta de recursos económicos para llevar a cabo investigación.

Esta tesis es un instrumento, que junto con otras fuentes de información, pretende impulsar el uso y desarrollo de CAMEX, explicando su funcionamiento y mostrando la forma de llevar a cabo experimentos mediante la modificación de su código fuente y su interfaz de usuario.

Desarrollos futuros

Es bueno también señalar que dadas sus características, CAMEX aún puede seguir siendo ampliado y mejorado, presentándose como una buena opción para el desarrollo profesional de personas interesadas en programación, fun-

cionamiento de *hardware*, y por supuesto, autómatas celulares. Incluso, CAMEX puede ser una buena base para la creación de un programa que controle nuevas versiones de máquinas de autómatas celulares, como la CAM-8, el modelo más reciente de estos instrumentos desarrollado por el MIT. A continuación se expone una serie de cosas que los autores consideran serían útiles y que podrían desarrollarse a corto plazo en futuros trabajos.

Pulir el código. El programa CAMEX está escrito en un estilo no conforme a los estándares actuales de programación en lenguaje C de la ANSI. Casi cualquier compilador actual de C desplegará mensajes de advertencia (*warnings*) por la forma en que está escrito el programa (principalmente por la forma en que se declaran las funciones), por lo que hay que deshabilitar los mensajes de error del compilador. Esto podría evitarse rescribiendo el programa en una forma apegada a los estándares actuales.

Implementación de un ratón Aunque CAMEX tiene de hecho los elementos para la incorporación de un ratón (*mouse*), en la época en que CAMEX fue escrito los ratones no eran un accesorio estándar de las computadoras personales, por lo que se necesitaba un controlador especial para estos dispositivos. Puede ajustarse CAMEX para reconocer un ratón cuando se carga el programa, y modificar varios aspectos de CAMEX para sacar provecho de este dispositivo.

Interfaz gráfica El manejo de CAMEX puede hacerse más intuitivo programando una interfaz gráfica con los elementos característicos de éstas, como las ventanas, barras de menús y los menús colgantes.

Guardar patrones Puede escribirse un programa para guardar la imagen que se encuentre en el Buffer de Planos (mediante un operador de REC por ejemplo) en un archivo con un formato común (como un mapa de colores).

Manual Estre trabajo explora sólo las principales facetas que deben conocerse de CAMEX (y de la CAM-PC) para programar dispositivos CAM. Es deseable un Manual exhaustivo que documente todos los aspectos del programa.

Pasado y futuro de la CAM

Las computadoras de propósito específico CAM, surgieron debido a la necesidad de un simulador de autómatas celulares capaz de ejecutar con rapidez el gran número de cálculos que requieren estos modelos cuando se trabaja en dos dimensiones. La CAM fue concebida y desarrollada por Tommaso Toffoli en 1981, inicialmente como un proyecto personal; la versión CAM 1.2 fue construida dentro de la investigación del MIT, con financiamiento parcial por este instituto. En 1984, Toffoli publicó un artículo en *Physica D*sobre la CAM, donde expresaba su deseo de que varios grupos de investigación tuvieran la posibilidad de poseer su propia copia de la máquina para realizar sus estudios. En ese entonces los recursos tecnológicos constituían un impedimento para tener vecindades preconstruidas, por lo que las líneas de entrada y salida de las tablas de consulta tenían que ser manipuladas manualmente por medio de cables para obtener el tipo de vecindad deseado; como consecuencia el manejo de la tarjeta únicamente con *software* no era viable.

La versión llamada CAM-6 fue disponible comercialmente a la comunidad científica al ser producida por la compañía SYSTEMS CONCEPTS, de San Francisco, California. A partir de esta versión de la CAM surgió la computadora CAM-PC (producida por Automatrix, Inc.), la cual era virtualmente muy parecida a la CAM-6, pero tecnológicamente su construcción era mucho más avanzada.

Actualmente se encuentra en el mercado la versión CAM-8, la cual es superior a la CAM-PC y se distribuye con una estación de trabajo SUN. Esta nueva herramienta seguramente tendrá un papel muy importante en las investigaciones sobre autómatas celulares.

Apéndice A

Información adicional sobre CAMEX

A.1 Cómo compilar CAMEX

El programa CAMEX consta de los siguientes archivos:

```
CAM.C
CAMEML.C
CAMETD.C
CAMEWW.C
CAMPC.H
CAMTA.C
CAMTBL.H
REC.H
CAMD.C
CAMEMT.C
CAMEV.C
CAMEX.H
CAMPL.C
CAMTAM.C
CAMUV.C
RUIN21.C
CAMEHT.C
CAMEQT.C
CAMEVT.C
```

```
CAMOD.C
CAMSU.C
CAMTAV.C
KEYBRD.H
RUIN4H.C
VIDEOHID.OBJ
TMREC.LIB
```

Aparte de los archivos anteriores, que conforman el programa y deben ser compilados juntos, existen otros archivos con extensión .PAT, que son simplemente patrones que pueden ser leídos desde un disco y escribirse como una configuración en el Buffer de Planos.

CAMEX fue compilado con la versión 2.01 del Turbo C de Borland, utilizando el modelo de memoria *medio* y deshabilitando todos los mensajes de advertencia que da el compilador. Para compilar CAMEX es necesario compilar simultáneamente (con la opción de *proyecto*) todos los archivos de la lista anterior.

Bibliografía

[1] Berlekamp, Conway y Guy. *Winning Ways.* Academic Press, 1982.

[2] Andrea Califano, Norman Margolus, and Tommaso Toffoli. CAM-PC: *A High-Performance Cellular Automata Machine USER'S GUIDE.* MIT Laboratory for Computer Science, Cambridge, MA, revision 0.1 edition, 1990.

[3] Edgar F. Codd. *Cellular Automata.* Academic Press, Nueva York, NY, 1968.

[4] A. K. Dewdney. Juegos de ordenador. el juego vida cuenta ya con sucesores en tres dimensiones. *Investigación y Ciencia No. 27, pp. 94-99,* 1987.

[5] Martin Gardner. Mathematical games - the fantastic combinations of john h. conway's new solitaire game life. *Scientific American,* 1969.

[6] Martin Gardner. *Wheels, Life and other Mathematical Amusements.* W. H. Freeman and Company, 1983.

[7] Howard Gutowitz. *Cellular Automata, Theory and Experiment.* MIT Press, Cambridge, MA, 1991.

[8] Erica Jen. Aperiodicity in one-dimensional cellular automata. In Howard Gutowitz, editor, *Cellular Automata, Theory and Experiment.* MIT Press, 1991.

[9] Harold V. McIntosh. *Linear Cellular Automata.* Universidad Autónoma de Puebla, Apartado Postal 461 (72000) Puebla, Puebla, México, 1990.

[10] Harold V. McIntosh. *What Has and What Hasn't Been Done With Cellular Automata.* Universidad Autónoma de Puebla, Apartado Postal 461 (72000) Puebla, Puebla, México, 1990.

[11] Harold V. McIntosh. *Linear Cellular Automata via de Bruijn Diagrams.* Universidad Autónoma de Puebla, Apartado Postal 461 (72000) Puebla, Puebla, México, 1991.

[12] Harold V. McIntosh. *The* CAM/PC *exerciser* CAMEX. Universidad Autónoma de Puebla, Apartado Postal 461 (72000) Puebla, Puebla, México, 1992.

[13] MIT Laboratory for Computer Science. *CAM-PC: A High-Performance Cellular Automata Machine Hardware Manual*, version 1.0 edition, 1990.

[14] Cosma Rohilla Shalizi. Notebooks. http://physserv1.physics.wisc.edu/ shalizi/notebooks.html.

[15] Ellen Thro. *Artificial Life Explorer's kit.* SAMS Publishing, Carmel, IN, 1993.

[16] Tomasso Toffoli. Cam: A high-performance cellular-automaton machine. *Physica10D*, 1984.

[17] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines: A new Environment for Modeling.* MIT Press, 1987.

[18] John von Neumann. *Cellular Automata, Theory and Experiment.* University of Illinois Press, Urbana, Ill, 1966. editado y completado por Arthur W. Burks.

[19] Fred Warren. The F-PC homepage. http://www.spiritone.com/ fwarren/fpc.html.

[20] Jörg Weimar. Simulation with cellular automata. http://www.tu-bs.de/institute/WiR/weimar/ZAscript/.

[21] Stephen Wolfram. *Cellular Automata and Complexity.* Addison Wesley, 1994.

[22] Gerardo Cisneros y Harold V. McIntosh. *Notas sobre los lenguajes REC y Convert.* Universidad Autónoma de Puebla, 1986.