



Centro de Experimentación e Investigación en Artes Electrónicas

Universidad Nacional de Tres de Febrero

Proyecto de investigación “Autómatas Musicales Interactivos” en el marco del CEI ArtE - Centro de Experimentación e Investigación en Artes Electrónicas - Universidad Nacional de Tres de Febrero (Argentina)

# AUTÓMATAS MUSICALES INTERACTIVOS

Investigadores: Matías Romero Costas, Emiliano Causa,  
Tarcisio Lucas Pirotta ([Grupo Biopus](#))

## Palabras clave

Autómatas celulares, composición algorítmica, composición asistida, vida artificial, sistemas multi-agentes, autorregulación, autoorganización, emergencia.

## 1. Introducción

Este trabajo forma parte del proyecto de investigación “Autómatas Musicales Interactivos” que se desarrolla dentro del marco del CEIArtE (Centro de Experimentación e Investigación en Artes Electrónicas), de la Universidad Nacional de Tres de Febrero.

El objetivo de esta investigación es indagar sobre la generación de estructuras musicales auto-organizadas a partir de la utilización de autómatas celulares en tiempo real, para su posible aplicación a sistemas multimedia interactivos que, a partir de la captación de movimiento (y otras formas de interacción), permitan generar una estructura musical auto-organizada cuyos parámetros sean capaces de adaptarse al entorno (principalmente a las decisiones de los usuarios), y evolucionen en el tiempo, a partir de transformar sus parámetros internos en diversos niveles de complejidad sonoro/musical. Los modelos de auto-organización que se estudiarán son principalmente los autómatas celulares.

El presente artículo es la continuación del texto “Autómatas celulares y música”, y muestra los nuevos avances en esta investigación.

En la primera parte abordaremos la problemática de la estructuración musical y su relación con los sistemas de complejos y de vida artificial, como son los autómatas celulares. Luego, en la segunda parte, explicaremos la arquitectura de las aplicaciones y los algoritmos utilizados en la programación del autómata celular, para finalmente concentrarnos con un ejemplo concreto donde se pone en funcionamiento el autómata celular para controlar la evolución del ritmo, la dinámica y la duración de los sonidos.

### 1.1. Alcances

En la presente investigación se articulan algoritmos musicales a partir de algoritmos de autómatas celulares, la vez que se vinculan ambos con sistemas de pantallas sensibles al tacto. El alcance de esta investigación se restringe a las problemáticas relacionadas con la aplicación de los algoritmos<sup>1</sup> musicales a los autómatas celulares, como una forma de estudiar la utilización de estos sistemas complejos en la creación musical. Los algoritmos musicales que se utilizan en esta investigación, así como los algoritmos para el desarrollo de pantallas sensibles<sup>2</sup> al tacto no son producto de esta investigación, y por ende escapan al alcance de la misma.

---

1 Los algoritmos de composición musical son de desarrollo propio, y algunos criterios de modularización se inspiran en el trabajo de Karlheinz Essl, quien construyó la librería “Real Time Composition Library”, de la que también fueron tomados algunos objetos

2 Algunos de ellos son de desarrollo propio y otro utilizando tecnologías existentes, como ReacTiVision.

## 1.2. Estructuración musical

Podemos definir a nuestro autómata como un algoritmo de composición musical que se vale de un modelo matemático de simulación de sistemas complejos autorregulados como medio para lograr una autoorganización musical en el tiempo. La elección de los autómatas celulares como modelo se debe a que, de la interrelación de sus partes constitutivas, emergen estados de organización de nivel superior.

Una particularidad de nuestro autómata celular, es que cada célula constituye, en realidad, un conjunto de parámetros y/o comportamientos musicales complejos, que son los que van a evolucionar a partir de la puesta en funcionamiento del autómata. Podría hacerse una analogía, solo a modo de ejemplificación, entre cada una de estas células como una de las líneas que conforman una polifonía.

De esta manera la célula es, en realidad, una unidad de generación musical, una entidad autónoma definida por una serie de parámetros y funciones musicales (como el ritmo, la dinámica y las alturas) que evolucionan, por un lado de forma independiente, y por otro, en función de la relación con sus vecinos más próximos.

Esta forma de entender a la célula se aleja de las concepciones tradicionales, donde cada célula solo representa el valor de un único parámetro (generalmente un número entero), y se ubica en un nivel de abstracción superior, más cercano a los mecanismos de la composición musical, donde un conjunto de parámetros musicales son puestos en funcionamiento de forma simultánea y entrelazada. Sin embargo, cabe aclarar que matemáticamente es posible transformar un conjunto de valores numéricos (parámetros en este caso) en un único número, mediante alguna regla de codificación. Si bien para nuestro caso no tiene sentido hacer esa transformación, dado lo engorroso del procedimiento, dicha posibilidad nos permite asegurar de que nuestro modelo no rompe con el paradigma del autómata celular.

Tomando a las células como unidades generadoras, el tipo de relación entre ellas queda establecido por todos los parámetros musicales puestos en juego. Este lazo puede producirse de dos maneras:

- entre parámetros comunes, donde por ejemplo el campo armónico de la célula afecta la de sus vecinas.
- entre diferentes parámetros, por ejemplo si la densidad cronométrica determina la dinámica y duraciones de otras células.

### 1.3. Relación entre la estructuración musical y los autómatas celulares

*“Una de las cuestiones más complejas con la que nos enfrentamos fue con la de generar un discurso estético a partir de la utilización de sistemas de simulación de vida artificial. La complejidad de la cuestión reside principalmente en lo difícil que es encontrar criterios que permitan articular discursos como el musical, a estructuras que evolucionan en el tiempo, en forma espontánea e imprevisible. Por*

*otra parte, gran parte de los sistemas de simulación relacionados con la vida artificial, generan sistemas complejos que aumentan aún más este nivel de impredecibilidad. Generalmente, vincular directamente, e irreflexivamente, la música y la imagen con la interactividad, trae resultados que, en el mejor de los casos impacta, pero que rápidamente cansan, por la falta de coherencia discursiva” (13. Causa y Romero Costas).*

La primera etapa de esta investigación arrojó luz sobre algunas cuestiones relacionadas con la forma de implementar los autómatas celulares para lograr una organización musical coherente, a saber:

La propiedad de “emergencia” se vuelve mas evidente a la percepción en autómatas celulares de grandes dimensiones, donde estados de orden de tipo orgánico surgen con mayor facilidad, sin embargo esto no se produce en autómatas de dimensiones pequeñas. Entonces, si bien es preferible trabajar con autómatas de mayor cantidad de células, aplicados a la música provocan, a la percepción, una falta de organización o un comportamiento de tipo aleatorio, debido a la saturación por la sumatoria de elementos que suenan al mismo tiempo. Este comportamiento puede modificarse si las células, más allá de la evolución interna de sus parámetros, suenan por turnos asignados por alguna función externa al autómata, en lugar de hacerlo constantemente.

En relación a lo anterior, cuando todas las células del autómata presentan el mismo comportamiento a lo largo del tiempo, resulta poco probable que surja una organización con coherencia musical. Esto podría subsanarse asignando funciones musicales particulares a regiones diferentes del autómata. Entonces la evolución de un determinado parámetro musical esta a cargo, solamente, de un grupo de células, en relación a sus vecinos, y no de todo el autómata.

Otra característica fundamental es que el desarrollo del autómata puede ser alterado por agentes externos, en este caso la intervención de un usuario, con la capacidad de desviar o modificar la evolución musical. Esto supone la inclusión de un parámetro subjetivo al devenir de dicha organización, que puede provenir de la simple experimentación sensible por parte del usuario, hasta decisiones estéticas y/o compositivas, en el caso, por ejemplo, de una persona con experiencia musical.

Estos argumentos definieron un modo diferente de abordar el problema, con una base en la percepción a nivel global de la evolución musical, con el objetivo de *“extraer perceptivamente los movimientos que producen los fenómenos emergentes, y usarlos como guía de la conducción de los parámetros musicales, apuntando a producir un movimiento análogo.” (Causa, Romero Costas)*

#### 1.4. Diferencias entre la estructuración musical y visual

Es evidente que existen diferencias en la forma de articular y estructurar la imagen y la música, una de ellas, quizás la más importante, es el tiempo. Mientras que la música expresa su estructura a lo largo del tiempo, la imagen lo hace, de forma preponderante, en el espacio. En nuestro caso la imagen es una representación gráfica de la evolución del autómatas, y debe dar cuenta lo mejor posible del estado de las células en cada momento.

Ahora bien, el problema reside en que cada célula está conformada por múltiples parámetros que pueden estar cambiando simultáneamente. Resulta sumamente difícil, sino imposible, representar en un espacio bidimensional, toda la complejidad presente a nivel musical. Una vez más optamos por representar solo aquellos rasgos que son más relevantes a la percepción y que dan cuenta de los cambios musicales más relevantes a nivel estructural.

La figura 1 muestra la representación gráfica del autómatas, donde cada uno de los círculos es una célula.

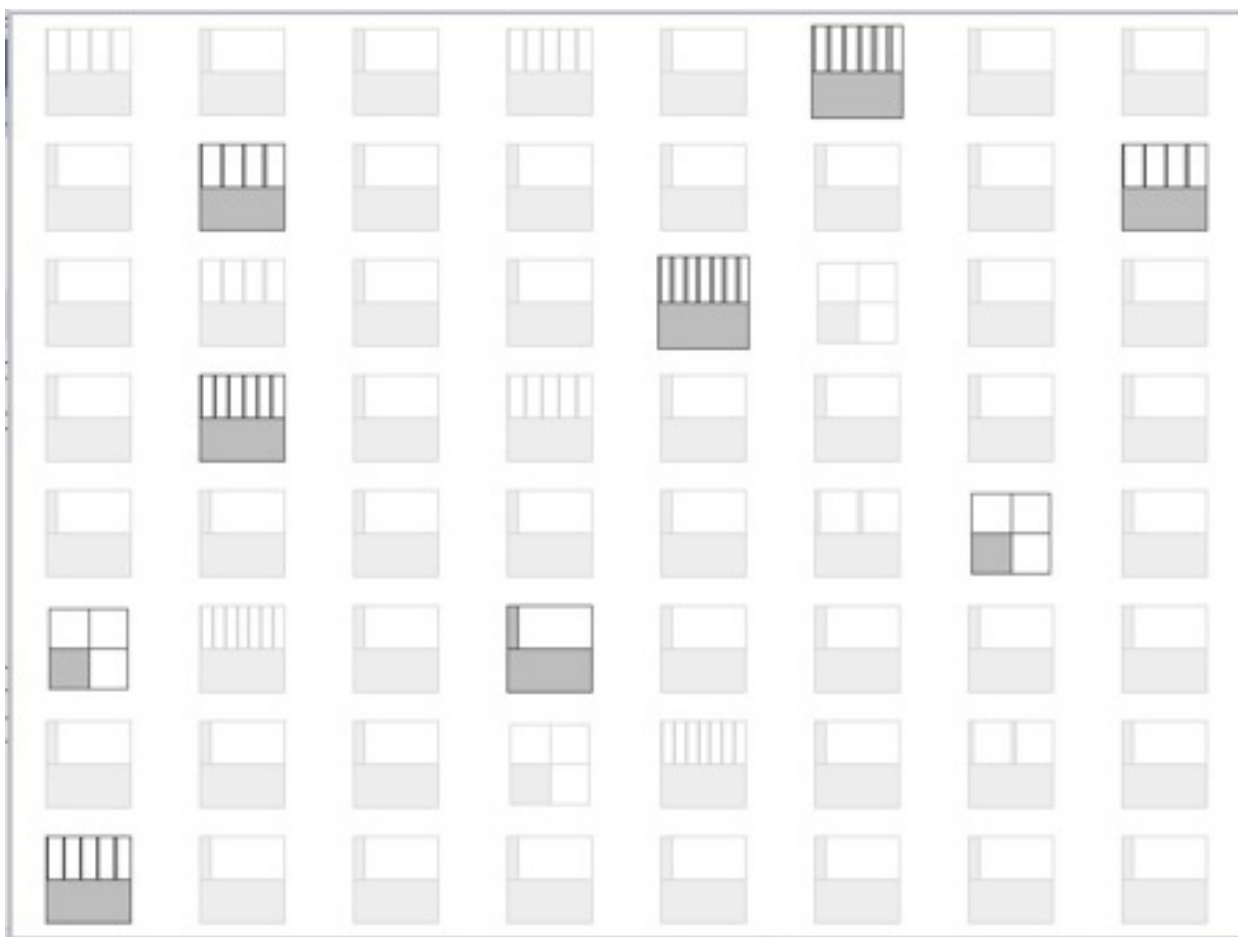


Figura 1: Interfaz gráfica del autómatas musical

Existen tres niveles de representación, asociados a tres niveles de pregnancia perceptual:

1. El valor de gris
2. La ubicación espacial
3. El contenido

En un primer nivel, cuando una célula está “activa”, es decir que está sonando, aparece representada con un valor de gris más oscuro, a diferencia de cuando está inactiva, que es representada en gris claro.

Su ubicación en el eje “y” (vertical) está asociada con el registro, así, las células ubicadas en la primera fila están en el registro más agudo, y las de más abajo en el más grave. A su vez cada columna representa un timbre (instrumento musical) diferente.

Finalmente, dentro de cada cuadrado, una serie de parámetros, como las figuras rítmicas, los silencios, la duración, etc., son representados a través de subdivisiones, líneas divisorias, líneas más gruesas, segmentos, etc.

Esta segmentación en niveles de representación de acuerdo con los niveles de percepción nos parece fundamental para una mejor comprensión de los procesos que se llevan a cabo internamente en el autómata y que ayudan, en consecuencia, a poder manipularlos.

## 1.5. Processing vs. Max

Se eligieron dos entornos de programación para el desarrollo de los algoritmos de composición musical, la representación visual y la vinculación con las interfaces tangibles: Processing <sup>1</sup> y Max/MSP <sup>2</sup>.

En base a la experiencia con estos dos entornos y lenguajes de programación hemos llegado a una serie de conclusiones:

Un lenguaje de programación por código, como Processing, es más adecuado para la programación de lógicas secuenciales, cosa que resulta sumamente engorrosa en lenguajes que trabajan en paralelo (entornos de programación visual por objetos), como Max/MSP.

- Max/MSP fue creado inicialmente como una herramienta de composición algorítmica, y por ello tiene sumamente desarrollado funciones de ejecución y generación musical a través del protocolo MIDI, lo que lo convierte en la mejor elección para el manejo de las funciones musicales del autómata.

---

1 Processing ([www.processing.org](http://www.processing.org)), entorno de programación basado en el lenguaje de programación Java, desarrollado en el MIT como una herramienta para artistas digitales.

2 Max-MSP-Jitter. (Miller Puckette, David Zicarelli y otros) Creada en el IRCAM (<http://www.ircam.fr/>), y desarrollada actualmente por Cycling74 ([www.cycling74.com](http://www.cycling74.com)).

- Debido a que Max/MSP/jitter tiene evidentes pérdidas de rendimiento en la visualización del comportamiento de autómatas celulares de dimensiones mayores a 20x20, se optó por la implementación de la imagen en Processing.
- Processing permite una vinculación más sencilla con dispositivos tangibles, como las pantallas sensibles al tacto.

Las características observadas y las comparaciones efectuadas entre Max/MSP y Processing permiten determinar fortalezas y debilidades de ambas plataformas para llevar a cabo tareas específicas, y determinaron, en parte, la arquitectura de la implementación del “autómata musical”, que abordaremos más adelante.

## 2. Programación del Autómata Musical

### 2.1. Distribución de tareas

En función de la complejidad de la programación, y siguiendo las características de los programas informáticos detallados anteriormente, se decidió implementar el autómata musical separándolo en tres aplicaciones, cada una encargada de una tarea específica.

- El instrumentista, programado en el entorno Max/MSP.
- La orquesta, utilizando el software Reason<sup>3</sup>
- El autómata, en el entorno de programación Processing

El **autómata** es el encargado de la evolución del autómata celular, sus reglas de comportamiento, el estado de cada célula, y las relaciones de vecindad. Por esa razón es el que tiene a su cargo las decisiones compositivas. También es el encargado de la representación gráfica del autómata.

El **instrumentista** recibe las ordenes del autómata y ejecuta los eventos musicales y configura el discurso musical en forma de eventos MIDI, que son enviados, a través de un puerto virtual, a la orquesta, que asigna a cada una de estas notas un timbre particular, asociado a un instrumento.

En el caso de querer manipular al “autómata musical” con una interfaz tangible, por caso una pantalla sensible al tacto, la aplicación de Processing puede recibir datos de control desde un programa externo vía protocolo de red.

---

<sup>3</sup> <http://www.propellerheads.se/>

## 2.1. Diseño de objetos y su vinculación

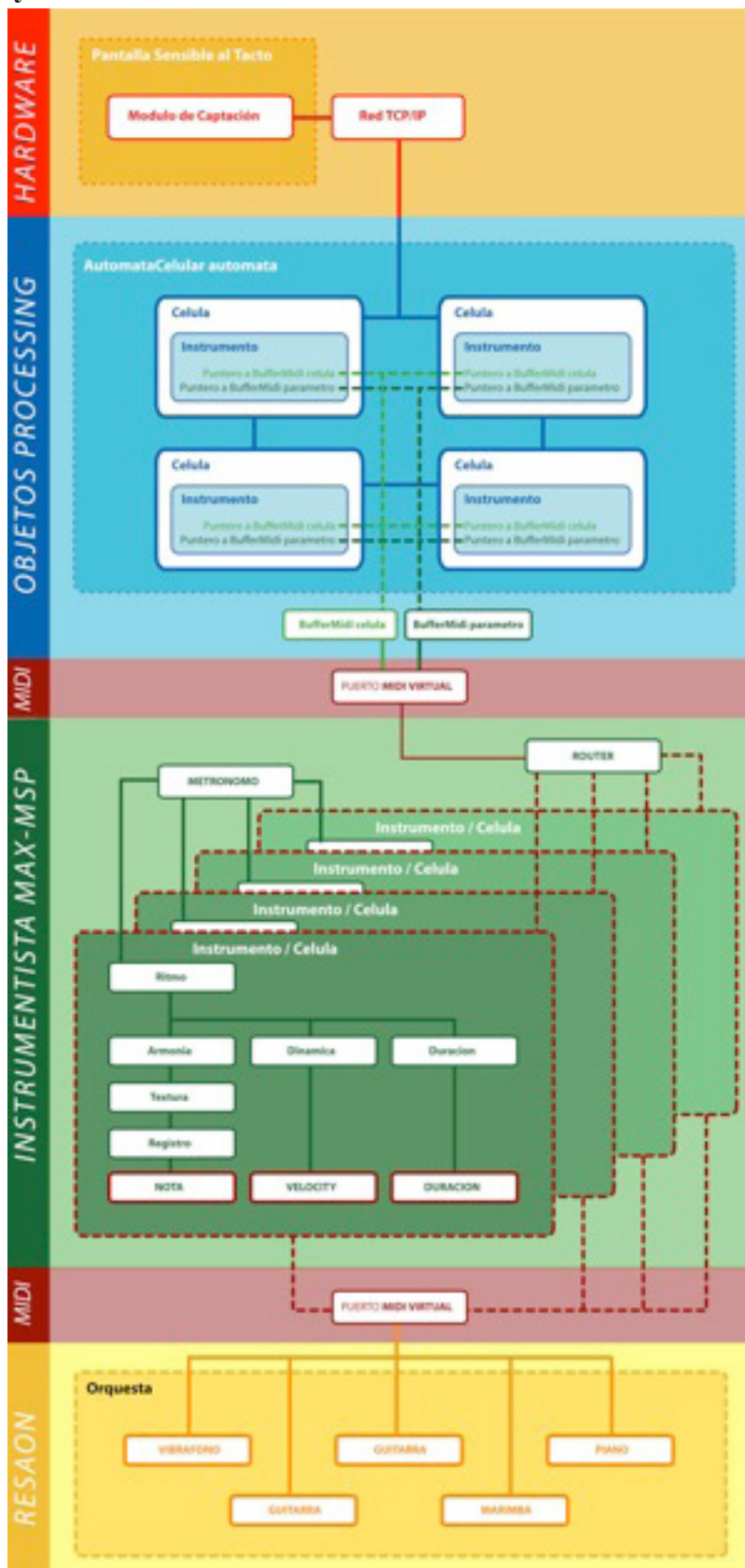


Figura 2: Estructura y conexión entre las partes del autómata musical

### 3. El Instrumentista

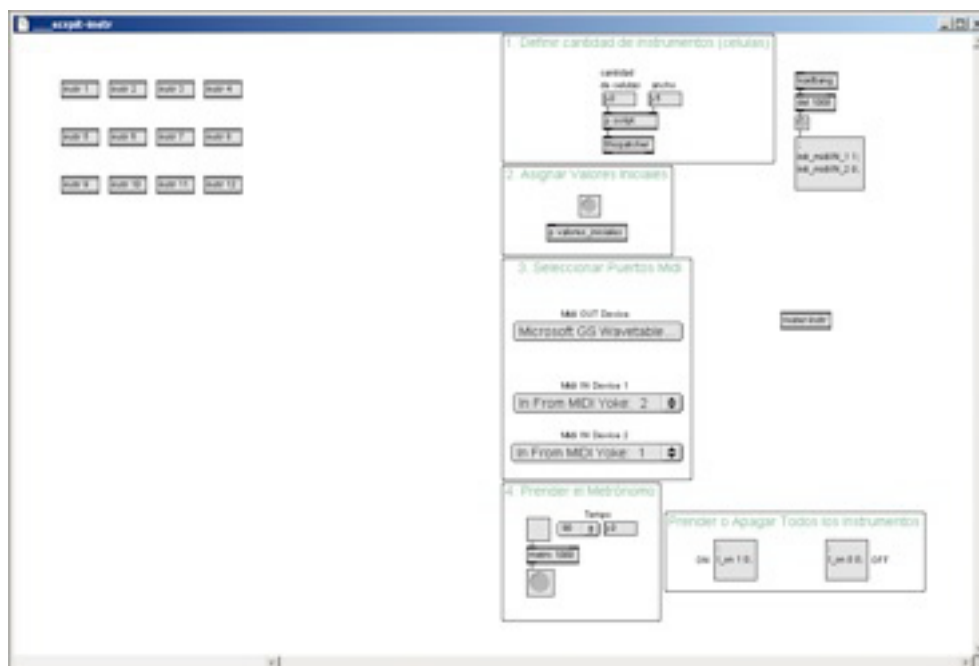


Figura 3: Pantalla principal del programa “Instrumentista” en Max MSP

Este programa permite crear de forma automática una cantidad N de instrumentos o células musicales, cada una de las cuales constituye una unidad de generación musical (UGM) controlada por un metrónomo general desde el programa principal, común a todas las células. Cada UGM está compuesta por módulos que tienen a su cargo el control de diferentes funciones musicales:

- Generador de ritmo
- Generador de Armonía
- Unidad de Control de la Dinámica
- Unidad de Control de la Duración

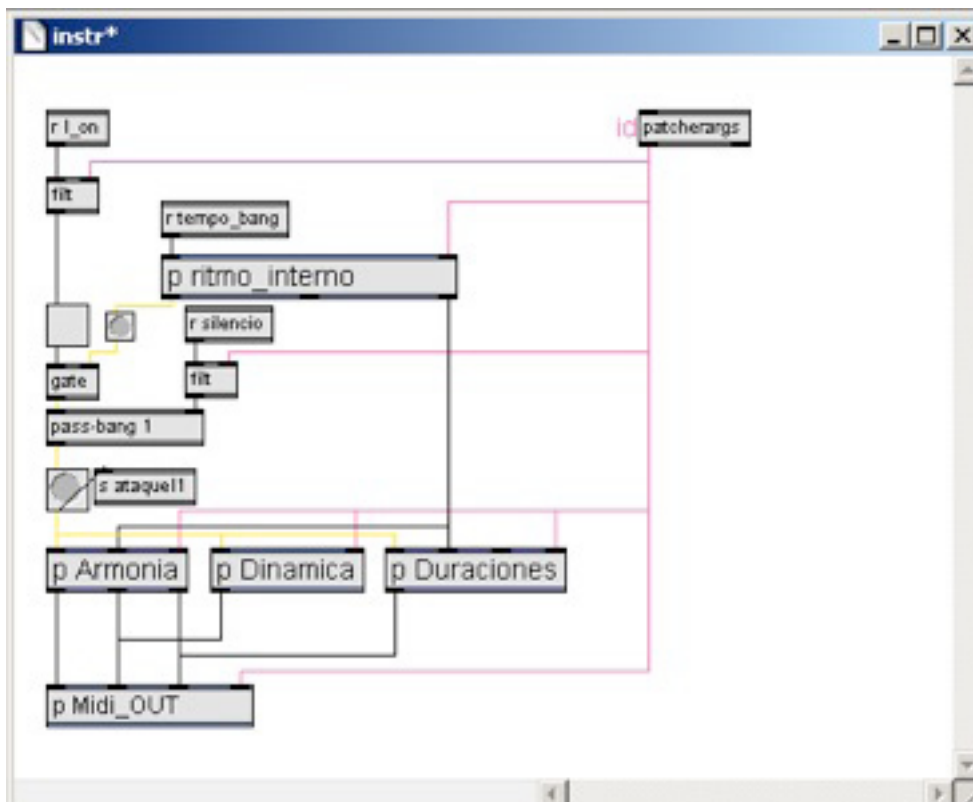


Figura 4: Pantalla principal del programa “Instrumento” en Max MSP

### 3.1.1. Unidad generadora de ritmo (UGR)

El generador de ritmo es el encargado de la organización rítmica de los eventos musicales, es decir su aparición y distribución en el tiempo. Todos los módulos siguientes dependen de este, y su función consiste en asignar una altura, una duración y una intensidad a cada uno de los pulsos generados por la UGR.

Esta unidad trabaja a partir de un pulso o metro, cuya velocidad es controlada por un metrónomo común a todas las células, y la generación de pulsaciones en forma de multiplicaciones o divisiones de dicho pulso. De esta manera se pueden crear figuras rítmicas que van desde la redonda a la semifusa, pasando por valores regulares e irregulares.

Por ejemplo si se quiere crear un tresillo de semicorchea, hay que dividir el metro en 3. Esta división establece un intervalo de tiempo entre los 3 ataques, igual a un tercio del tiempo del metrónomo. Si el metrónomo es igual a 60 (un pulso por segundo), el intervalo de tiempo entre las pulsaciones del tresillo será igual 1/3 de segundo.

Para evitar la regularidad absoluta de los procesos simulados por el programa, y para simular la “imperfeción” de una ejecución real (hecha por un ser humano) existe un parámetro denominado “rubato <sup>4</sup>”, que establece un grado de desviación temporal de entre 0 y 100 % de la pulsación de cada ataque.

<sup>4</sup> Rubato: tempo elástico, flexible, que permite ligeros acelerandos y ritardandos, de acuerdo con la demanda de la expresión musical (Diccionario Harvard de la Música).



información sobre los PCS véase 11 y 12). En segundo lugar se fija un tipo de configuración textural, e decir un plano sonoro, su nivel de variación y pregnancia, y su comportamiento en el tiempo. Configuraciones de tipo melódico, ostinatos, trinos, o trémolos pueden elegirse.

Finalmente el PCS es ubicado en un registro u octava, y puede permanecer fijo o puede moverse de manera ascendente, descendente o aleatoria.

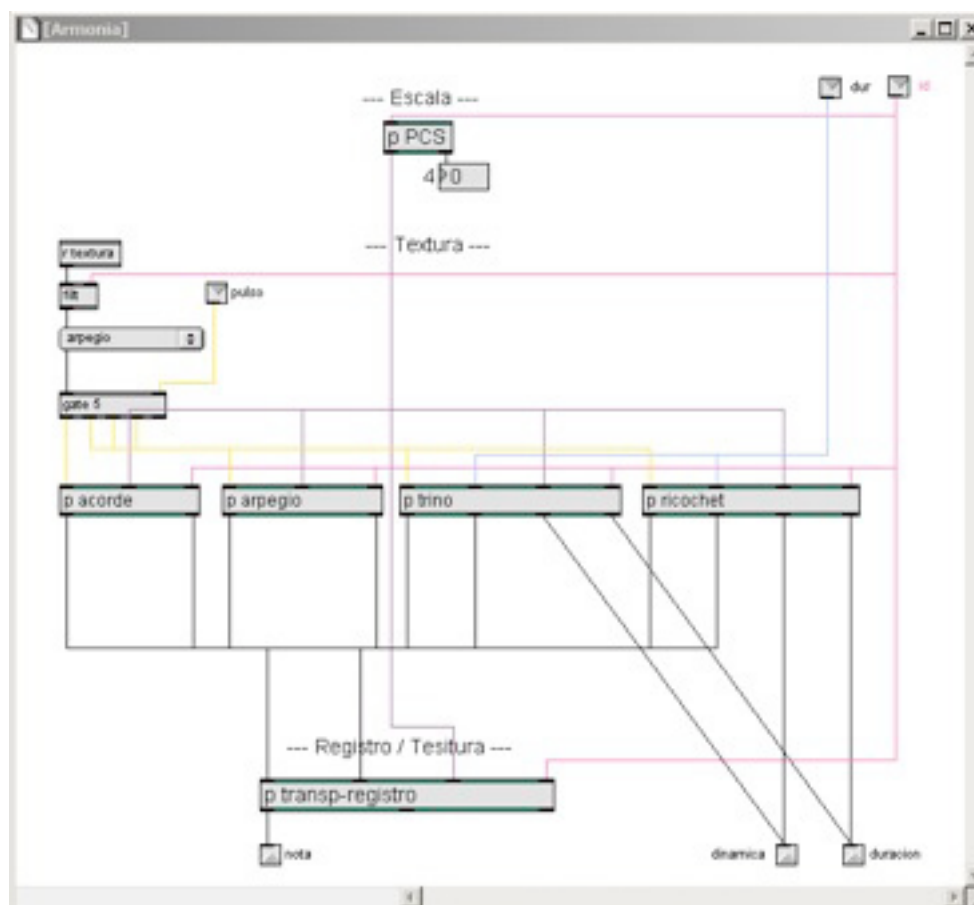


Figura 6: Pantalla de la UG “Armonía”

## Acorde

Esta función musical recibe el Pitch Class Set en su forma prima y lo distribuye verticalmente en el registro para ser ejecutados simultáneamente con cada ataque, y así generar una acorde. Un intervalo temporal en milisegundos permite controlar la entrada de cada una de las 4 alturas, lo que permite “arpeggiar” las notas. En el caso en que este intervalo es igual a 0 las 4 notas se ejecutan sincrónicamente, como en un acorde plaqué.

Para manejar las notas musicales en el programa utilizamos una representación numérica de las 12 alturas del total cromático:



Tomemos como ejemplo el PCS 4-26, cuya forma prima es 0 3 5 8, que pasado a alturas se corresponde con: Do-mib(re#)-Fa-Lab(sol#).



Una de las operaciones posibles sobre este conjunto es modificar su ordenamiento, lo que en matemática se conoce como permutación. Con un conjunto de 4 elementos podemos obtener 24 permutaciones<sup>5</sup>. Sin importar la nota de la que se trata, podemos nombrar el número de orden de n elementos en una secuencia que va de 0 a n-1, en nuestro caso de 0 a 3, donde 0 es la primera posición y tres la última.

0 1 2 3; 0 1 3 2; 0 2 1 3; 0 2 3 1; 0 3 1 2; 0 3 2 1; 1 0 2 3; 1 0 3 2; 1 2 0 3; 1 2 3 0; ...etc.

Si para las diferentes permutaciones tomamos como regla ir de lo grave a lo agudo, obtendremos arreglos de notas que van desde acordes “cerrados”. dentro de una octava (como en la parte a de la figura de abajo), a configuraciones abiertas, que ocupan 3 oct:



## Arpeggio

Esta función simplemente ordena de forma secuencial o aleatoria el grupo de 4 alturas con cada pulso que recibe de la UGR.

## Trino

Esta textura ejecuta trinos<sup>6</sup> entre pares de notas del PCS. El trino puede ser ejecutado a una velocidad constante, o estableciendo una velocidad inicial y una final, para lograr efectos de “acelerando” o “desacelerando”. Lo mismo

<sup>5</sup> La cantidad de permutaciones de n elementos puede calcularse por:  $P_n = n! = 4 \times 3 \times 2 \times 1 = 24$

<sup>6</sup> Trino: ornamento musical que consiste en una rápida alternancia entre dos notas contiguas.

sucede con la dinámica, que puede controlarse como un “crescendo” o “decrescendo”.

## Trémolo

El Trémolo es una repetición rápida de una sola nota, y comparte las propiedades de velocidad y dinámica con la función anterior.

### 3.1.3. Unidad de control de la dinámica (UGD)

De la misma manera que se define un campo de alturas y una configuración textural, cada pulso del UG de Ritmo es asociado a un valor dinámico. La dinámica puede ser fija, o variable en el tiempo, con un valor inicial, uno final y un intervalo temporal que permite simular los crescendos y decrescendos de la música. Al mismo tiempo es posible alterar la acentuación dinámica de las notas por grupos simples o compuestos para establecer un tipo de rítmica aditiva. Acentuaciones ca

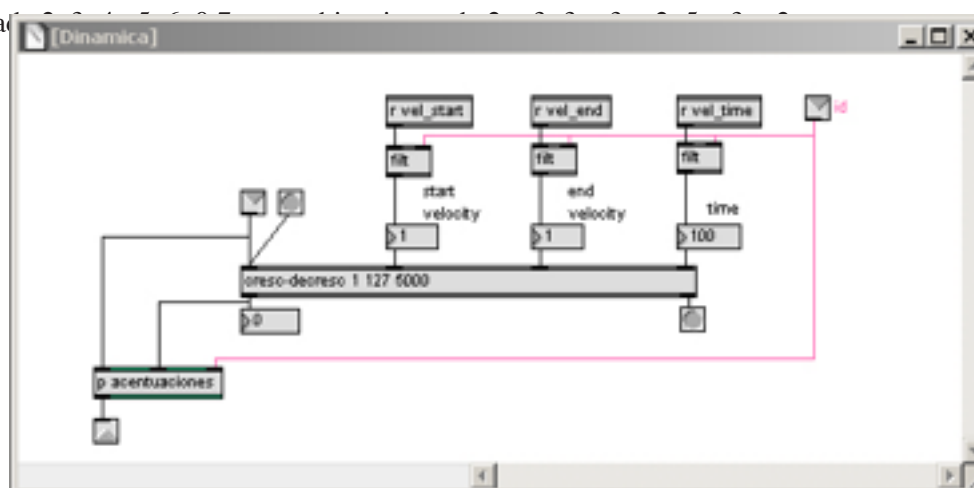


Figura 7: Pantalla de la UC “Dinámica”

### 3.1.4. Unidad de control de la duración (UCD)

En forma simultánea a la asignación de una dinámica se determina la duración de cada uno de las notas de forma independiente a los intervalos de ataque de cada pulsación. Si la duración de la nota se extiende más allá de la articulación de la próxima (la duración es mayor al intervalo de ataque) se puede simular la utilización del pedal en el piano. Es posible llegar al mismo resultado a través del controlador Midi correspondiente.

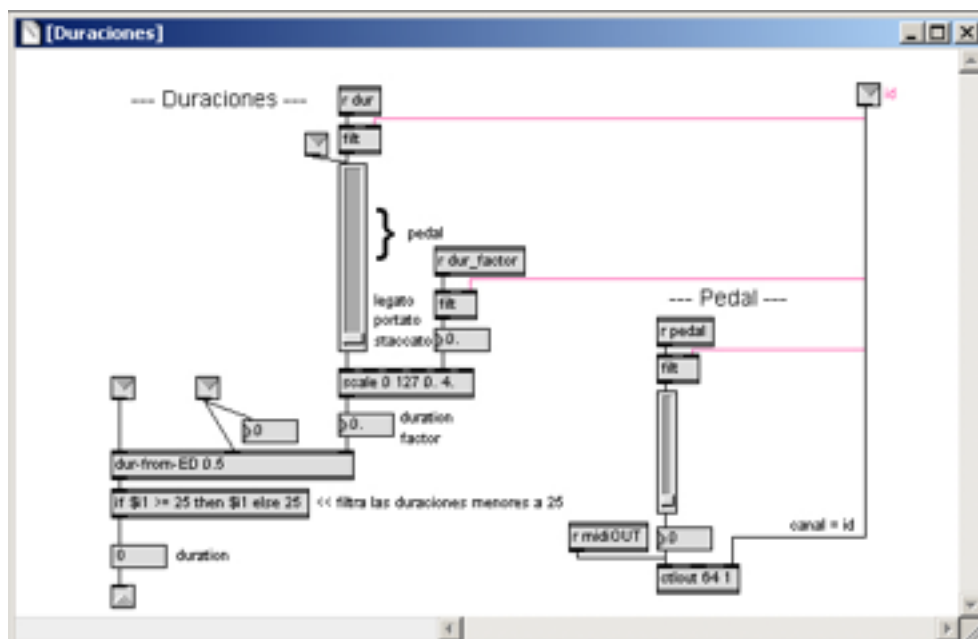


Figura 8: Pantalla de la UC “Duración”

#### 4. La orquesta

Este programa simplemente asocia a cada célula musical el timbre de un instrumento. Cada “instrumentista” tiene asignado un canal MIDI, y envía información a través de un puerto virtual hacia programa “Orquesta”. Es posible que varias células compartan el mismo canal, con lo cual serán asignados al mismo timbre. Esta decisión puede estar en relación a la organización textural de la música, ya que cada “instrumentista” puede definirse como un instrumento complejo, o bien como una línea simple de una configuración textural de nivel superior, como se verá en el caso de estudio explicado a continuación.

#### 5. El autómata

Este programa, implementado en Processing, tiene a su cargo la evolución del autómata celular; las decisiones “compositivas”, asociadas a dicha evolución; y la visualización de los procesos a lo largo del tiempo.

Para ello cuenta con una interfaz usuario que permite modificar, en tiempo real, cualquier parámetro musical de sus células, lo que agrega a su desarrollo autoorganizado la posibilidad intervenir y variar la evolución del autómata.

Debido a que uno de los objetivos de esta investigación es controlar los autómatas musicales a través de interfaces, el programa contempla el la manipulación de la interfaz usuario desde un programa externo, a través del protocolo de red TCP/IP, lo que permitiría controlarlo con un sistema de captura de movimiento, una pantalla sensible al tacto, o

cualquier otro sistema de registro de movimientos y gestos desarrollado para manipular la interfaz.

## 5.1. El Autómata Celular

En primer lugar hablaremos de la implementación de los algoritmos para la generación del autómata celular.

Para esto se ha definido la clase Autómata celular. Este autómata está formado por un vector de células. El constructor inicializa cada una de las células y al ser ejecutado recibe el ancho y el alto, cantidad de columnas y filas, y por último recibe dos objetos de tipo BufferMidi por referencia:

### Algoritmo 1: AutomataCelular (constructor)

```
AutomataCelular( int ancho , int alto , BufferMidi bufferCelula , BufferMidi bufferParametro , PFont fuente ){
    col = ancho;
    fil = alto;
    celulas = new Celula[col][fil];
    int idInstrumento = 1;
    for( int j=0 ; j<fil ; j++ ){
        for( int i=0 ; i<col ; i++ ){
            celulas[i][j] = new Celula( i , j , col , fil , bufferCelula , bufferParametro , idInstrumento , fuente );
            idInstrumento++;
        }
    }
    localizar(0,0,width,height);
}
```

Para que cada célula tenga su propio identificador, también recibe como parámetro, un valor de idInstrumento que se va incrementando.

Los dos objetos BufferMidi que recibe el autómata por referencia ( bufferCelula y BufferParametro) , que abordaremos más adelante con detenimiento al hablar del instrumento, son declarados al comienzo cuerpo principal del programa, cada uno corresponde a cada puerto MIDI y envían la información correspondiente al número de célula y el valor del parámetro que se quiere modificar respectivamente. En el mismo cuerpo principal del programa, dentro de la función miSetup() están declarados dos objetos midiIO, definiendo el puerto 2, para la célula y el 3 para los parámetros.

**Algoritmo 2: miSetup()**

```
void miSetup(){  
  
    midiIO = MidiIO.getInstance(this);  
    midiIO.printDevices();  
  
    int puerto = 2;  
    int canal = 1;  
    midiCelula = midiIO.getMidiOut( canal , puerto );  
  
    puerto = 3;  
    canal = 0;  
    midiParametro = midiIO.getMidiOut( canal , puerto );  
  
    bufferCelula = new BufferMidi( midiCelula , 100 , “celula” );  
    bufferParametro = new BufferMidi( midiParametro , 100 , “parametro” );  
  
    automata = new AutomataCelular( 4 , 3 , bufferCelula , bufferParametro , fuente );  
}
```

**5.2. Célula e Instrumento**

Como hemos mencionado, la aplicación que describimos está pensada como una interfaz, donde cada célula del autómata es utilizada para contener un instrumento.

Para implementar esta interfaz a través de la cual las células representan y controlan los “instrumentistas” en MAX/MSP, dentro de cada célula se declara internamente un objeto Instrumento, es decir, la célula en sí misma no es un instrumento, sino que es contenedora del mismo, el cual ha sido diseñado en una clase aparte.

**5.2.1. Célula**

La clase Célula se encarga de cosas generales, por ejemplo en el constructor recibe la información de su ubicación dentro de la matriz del autómata y la cantidad de filas y columnas que tiene el autómata , para así saber la vecindad con las demás células.

Recibe un idInstrumento y los buffer anteriormente ya mencionados, finalmente en la última línea de la clase se declara

el objeto Instrumento el cual contiene al instrumento propiamente dicho como su denominación indica.

### Algoritmo 3: Célula

```
class Celula{

    Instrumento instrumento;

    float x,y,ancho,alto;

    int vecinoX[],vecinoY[];
    BufferMidi bufferCelula;
    BufferMidi bufferParametro;

    Celula( int i , int j , int col , int fil , BufferMidi bufferCelula_ , BufferMidi bufferParametro_ , int idInstrumento , PFont fuente ){

        bufferCelula = bufferCelula_;
        bufferParametro = bufferParametro_;

        vecinoX = new int[8];
        vecinoY = new int[8];

        //n
        vecinoX[0] = i;
        vecinoY[0] = j-1;
        //ne
        vecinoX[1] = i+1;
        vecinoY[1] = j-1;
        //e
        vecinoX[2] = i+1;
        vecinoY[2] = j;
        //se
        vecinoX[3] = i+1;
        vecinoY[3] = j+1;
        //s
        vecinoX[4] = i;
```

```

vecinoY[4] = j+1;
//so
vecinoX[5] = i-1;
vecinoY[5] = j+1;
//o
vecinoX[6] = i-1;
vecinoY[6] = j;
//no
vecinoX[7] = i-1;
vecinoY[7] = j-1;

for( int k=0 ; k<8 ; k++ ){
    vecinoX[k] = ( vecinoX[k] < 0 ? col-1 : vecinoX[k] );
    vecinoX[k] = ( vecinoX[k] > col-1 ? 0 : vecinoX[k] );
    vecinoY[k] = ( vecinoY[k] < 0 ? fil-1 : vecinoY[k] );
    vecinoY[k] = ( vecinoY[k] > fil-1 ? 0 : vecinoY[k] );
}
instrumento = new Instrumento( idInstrumento , bufferCelula , bufferParametro , fuente );
}

```

### 5.2.2. Instrumento

A su vez los parámetros que mencionamos en la célula (idInstrumento y los buffer) son recibidos en el constructor de la clase Instrumento. El idInstrumento es necesario ya que cuando el instrumento envíe un mensaje MIDI para modificar algún parámetro, en primer lugar debe enviar el ID para identificar el instrumento de destino. Recordemos que este instrumento es un espejo del instrumento implementado en la otra plataforma, en este caso en Max/MSP, todo el sistema de los instrumentos funciona a modo de espejo.

#### Algoritmo 4: Parámetros que recibe el constructor de la clase Instrumento

```
Instrumento( int id_ , BufferMidi celula_ , BufferMidi parametro_ , PFont fuente_ )
```

El instrumento tiene por cada tipo de parámetro en Max, un objeto de tipo BarraOpciones, declarados desde el parámetro r\_on, r\_silencion hasta r\_pedal que es el último, un total de 37 objetos, los que se inicializan en el constructor del objeto instrumento.

**Algoritmo 5: Declaración de los 5 primeros objetos BarraOpciones**

```

I_on = new BarraOpciones( 1 , "I_on" , 0 , 1 , 0 , leftMenu+m , topMenu+m*4 , m*6 , m*1 , fuente , m);
r_silencio = new BarraOpciones( 2 , "r_silencio" , 2 , 6 , 2 , leftMenu+m , topMenu+m*6 , m*18 , m*1 , fuente , m);
rit_tempo = new BarraOpciones( 3 , "rit_tempo" , 0 , 15 , 6 , leftMenu+m , topMenu+m*8 , m*18 , m*1 , fuente , m);
rit_figura = new BarraOpciones( 4 , "rit_figura" , 0 , 10 , 5 , leftMenu+m , topMenu+m*10 , m*18 , m*1 , fuente , m);
rit_rubato = new BarraOpciones( 5 , "rit_rubato" , 0 , 100 , 4 , leftMenu+m , topMenu+m*12 , m*18 , m*1 , fuente , m);

```

Como vemos en el algoritmo 5, la clase BarraOpciones recibe los siguientes parámetros: número de parámetro, etiqueta del nombre, mínimo y máximo, valor por defecto. Estos valores deben coincidir con los definidos en la aplicación de Max/MSP. Finalmente recibe parámetros relacionados a la posición en pantalla: left, top, ancho, alto y el tamaño de la fuente. Estos últimos parámetros son operados por un módulo M para poder hacer ajustable la interfaz.

Volviendo a la estructura de la clase Instrumento, cada parámetro además de tener definido un objeto BarraOpciones, tiene definidos un comportamiento Set y un Get. Esto se debe a que si a nivel de código esto es modificado desde el autómata celular, se invoca al comportamiento get o set, por lo cual hay definidos en total 37 get y 37 set, uno para cada tipo de parámetro del instrumento:

**Algoritmo 6: Declaración del get y set del primer parámetro I\_on :**

```

void set_I_on( int valor ){
    I_on.set( valor );
    enviarParametros( I_on.id , I_on.valor );
}
int get_I_on(){
    return I_on.valor;
}

```

Los comportamientos de tipo set, por ejemplo `set_r_silencio( int valor )`, ejecutan a su vez un comportamiento set dentro de la clase `BarraOpciones`, enviándole el valor y luego ejecutando el envío de parámetros por MIDI al buffer correspondiente.

Precisamente para esto, los comportamientos set emplean el comportamiento `enviarParametros( int idParametro , int valor )`, también perteneciente a la clase `Instrumento`. Esta función recibe el id del parámetro y el valor correspondiente. En primer lugar envía un mensaje del controlador al buffer de la célula, por el controlador 2, y le pasa su ID de instrumento. Luego realiza algo similar enviando un mensaje de controlador al buffer del parámetro, con el id de parámetro y el valor del parámetro (como vimos anteriormente en la referencia del algoritmo 3).

#### Algoritmo 7: enviarParametros

```
void enviarParametros( int idParametro , int valor ){
    celula.agregarControlador( 2 , id );
    parametro.agregarControlador( idParametro , valor );
    if( monitor ){
        println(" envÃo celula "+id);
        println(" envÃo parametro "+idParametro+" -> "+valor);
    }
}
```

#### 5.2.2.1. BufferMidi

Los objetos `BufferMidi` sirven para almacenar los datos que envía cada célula y generar con estos una cola de mensajes, para evitar situaciones críticas en la comunicación de los datos a través del puerto MIDI.

La clase `BufferMidi` contiene tres vectores para almacenar el número, el valor del parámetro y el tipo de parámetro MIDI entendido como mensaje MIDI.

Si recibe un mensaje de controlador lo pone en la cola, y esto se realiza a través del comportamiento `agregarControlador( int numero_ , int valor_ )`. La función del comportamiento es agregar en los vectores del buffer el parámetro siguiente, mientras haya espacio dentro del espacio total que tiene el buffer.

### Algoritmo 8: agregarControlador

```
void agregarControlador( int numero_ , int valor_ ){
    if( cantidad<total ){
        tipo[ entrada ] = "controlador";
        numero[ entrada ] = numero_;
        valor[ entrada ] = valor_;
        entrada = (entrada+1) % total;
        cantidad++;
    }
}
```

Para controlar si se han enviado todos los mensajes disponibles, o si existen aún mensajes en la cola, se utiliza la función actualizar(), que es ejecutada una vez por fotograma dentro del cuerpo principal del programa. Específicamente revisa si hay objetos encolados y envía el mensaje MIDI que corresponde, haciéndolo de a uno por vez y por fotograma, para evitar congestión del puerto.

### Algoritmo 9: actualizar()

```
void actualizar(){
    if( cantidad>0 ){
        int actual = (frameCount%100);
        if( ultimoFrame != actual ){
            if( tipo[ salida ] == "controlador" ){
                midi.sendController( new Controller( numero[ salida ] , valor[ salida ] ) );
                if( monitor ){
                    print( " Buffer " + etiqueta + "[ " + frameCount + " ] ---> Controlador num = " );
                    println( numero[ salida ] + " val = " + valor[ salida ] );
                }
            }
        }

        salida = (salida+1) % total;
        cantidad--;

        ultimoFrame = actual;
    }
}
```

Luego de que el autómata fue declarado y de haber comenzado la aplicación, el comportamiento `miDraw`, dentro del cuerpo principal del programa, comienza a ejecutarse indefinidamente en cada fotograma mientras la aplicación este funcionando. Esto permite por ejemplo que estén constantemente actualizándose los buffer ( ver algoritmo 10 ), primero el buffer de las células y luego del de los parámetros. Luego ejecuta también los comportamientos de actualización e impresión del autómata.

**Algoritmo 10: actualización de los buffer en el cuerpo principal del programa.**

```
void miDraw(){  
  
    background( 255 );  
    fill( 50 );  
    bufferCelula.actualizar();  
    bufferParametro.actualizar();  
    automata.actualizar( punteros );  
    automata.imprimir();  
}
```

Como hemos mencionado, los buffer están declarados por fuera de los autómatas, cuando se declara el autómata, recibe como parámetro al constructor los dos buffer. De esta manera, al pasar como parámetros objetos que ya están declarados en el cuerpo principal del programa, el autómata celular los recibe como si fueran punteros (ver algoritmo 1).

Un punto clave para comprender del mecanismo del sistema, es tener en cuenta que cada buffer es enviado por parámetro a la declaración de cada una de las células, para que todas tengan referencia donde está ubicado el buffer, el cual es el mismo para todas las células. No está multiplicado, toda la información de las diferentes células es enviada al mismo buffer, y este la reenvía según el orden en que recibió los datos de cada célula

### 5.3. Interfaz

El instrumento en su constructor inicializa lo objetos `BarraOpciones`, recordemos uno para cada tipo de parámetro del instrumento, pasándole entre otras cosas los valores de mínimo, máximo y defecto (ver algoritmo 5).

Como también hemos mencionado anteriormente los parámetros pueden ser modificados a nivel de código, ejecutando los comportamientos de tipo set, por ejemplo `set_rit_tempo( int valor )` con su valor correspondiente.

Sin embargo la idea principal de la aplicación es ser utilizada como interfaz gráfica, incluso pudiéndose implementar a través de una pantalla sensible multi-táctil. Entonces para hacer esto posible se utiliza un menú.

El funcionamiento del menú depende principalmente del comportamiento ejecutarMenu, perteneciente a la clase Instrumento, y por otro lado de una clase denominada Puntero. La función void ejecutarMenu( Puntero este ) recibe como parámetro un objeto Puntero, el cual ha sido desarrollado para otra investigación y se relaciona con las pantallas sensibles. Como mencionamos si bien podría emplearse para pantallas sensibles, en este caso la misma estructura del algoritmo permite que pueda ser utilizado con los datos de posición del mouse directamente.

Al declararse un objeto Puntero, este recibe como parámetros un ID, la posición en x e y, y tres valores booleanos: activo, up y down. Luego estos serán los parámetros que reciba el comportamiento ejecutarMenu.

#### Algoritmo 11: clase Puntero

```
class Puntero{
  //etiqueta -17-

  int id;
  float x;
  float y;
  boolean activo;
  boolean down;
  boolean up;
  boolean actualizado;
  Puntero(){
    x = 0;
    y = 0;
    activo = false;
    down = false;
    up = false;
    actualizado = false;
  }
  void desactualizar(){
    actualizado = false;
  }
}
```

Dentro de la clase Instrumento, una variable paginaMenu, es utilizada para paginar el menú. Esta variable se modifica a través de tres objetos de tipo Boton, uno para cerrar el menú y otros dos para siguiente y anterior página.

El objeto Boton o los objetos BarraOpciones, tiene un comportamiento actualizar, que recibe como parámetro el objeto puntero. Según el valor de la variable paginaMenu se muestran (actualizan) unos determinados parámetros (BarraOpciones) en grupos de 5, es decir 5 por “página”.

**Algoritmo 12: actualización de los parámetros según la página, en este caso la primera página.**

```
if( paginaMenu == 0 ){
    if( I_on.actualizar( este ) ) enviarParametros( I_on.id , I_on.valor );
    if( r_silencio.actualizar( este ) ) enviarParametros( r_silencio.id , r_silencio.valor );
    if( rit_tempo.actualizar( este ) ) enviarParametros( rit_tempo.id , rit_tempo.valor );
    if( rit_figura.actualizar( este ) ) enviarParametros( rit_figura.id , rit_figura.valor );
    if( rit_rubato.actualizar( este ) ) enviarParametros( rit_rubato.id , rit_rubato.valor );
}
```

El objeto Boton recibe un objeto llamado mouse, de tipo Puntero, y se utiliza para controlar si el puntero del mouse se encuentra dentro del área del botón, si es así actualiza los datos correspondientes. Por ejemplo si el botón de encendido fue presionado, se ejecuta el comportamiento cerrarMenu de la clase Instrumento. En el caso de siguiente y anterior es similar, sólo que no ejecutan ningún comportamiento, solo modifican al variable paginaMenu incrementándola o disminuyéndola, modificando así la paginación como se explico anteriormente.

**Algoritmo 13: comportamiento actualizar del botón.**

```
void actualizar( Puntero mouse ){
    if(mouse.x > left && mouse.x < left+ancho && mouse.y > top && mouse.y < top+alto ){
        down = mouse.down;
        up = mouse.up;
        over = true;
        activo = mouse.activo;
    }else{
        activo = false;
        down = false;
        over = false;
    }
}
```

En el caso del objeto de BarraOpciones, en el comportamiento actualizar devuelve un valor booleano. Si el mouse está dentro de la región de la barra, calcula en qué punto de la barra está ubicado el puntero del mouse, establece el valor nuevo que debe tener el parámetro correspondiente a dicha barra de opciones. Si ese valor ha cambiado con respecto al valor anterior que tenía, devuelve el valor correspondiente indicando si el valor fue o no actualizado. Si el valor es true (actualizado), entonces además envía los nuevos datos del parámetro.

**Algoritmo 14: comportamiento actualizar del barraOpciones.**

```
boolean actualizar( Puntero mouse ){
    boolean devuelve = false;
    if( mouse.down ){
        if(mouse.x > left && mouse.x < left+ancho && mouse.y > top && mouse.y < top+alto ){
            float offset = mouse.x-left;
            int nuevo = int( offset / modulo ) + minimo;
            valor = int( constrain( nuevo , minimo , maximo ) );
            if( valor != antevalor ){
                devuelve = true;
                antevalor = valor;
            }
        }
    }
    return devuelve;
}
```

Para que todo el sistema de la interface funcione correctamente, por ejemplo permitiendo abrir un menú por vez y una vez que un menú fue abierto, deshabilitar los correspondientes a las demás células, siempre deben estar los datos actualizados y cada célula debe tener la información de si se ha abierto o no un menú, y a la vez ser “consciente” del estado de ejecución del sistema en general.

El comportamiento actualizar, de la clase AutomataCelular, ejecuta por cada célula un comportamiento operar (ver algoritmo 15 ). Este comportamiento realiza dentro de cada célula la actualización de la propia célula (ver algoritmo 16) a través de una tercera función actualizarCelula (ver algoritmo 17). La función Actualizar célula recibe el id de la célula propia y los datos de las ocho células vecinas, además del x e y.

**Algoritmo 15: comportamiento actualizar de la clase AutomataCelular .**

```

void actualizar( Punteros punteros ){

    for( int j=0 ; j<fil ; j++ ){
        for( int i=0 ; i<col ; i++ ){
            celulas[i][j].operar( this );
        }
    }
    for( int j=0 ; j<fil ; j++ ){
        for( int i=0 ; i<col ; i++ ){
            celulas[i][j].actualizar( punteros , menuHabilitado );
            if( celulas[i][j].instrumento.abrioMenu ) menuHabilitado = false;
            if( celulas[i][j].instrumento.cerroMenu ) menuHabilitado = true;
        }
    }
}

```

**Algoritmo 16: comportamiento operar de la clase celula.**

```

void operar( AutomataCelular todos ){

    actualizarCelula( instrumento
        , todos.celulas[ vecinoX[0] ][ vecinoY[0] ].instrumento
        , todos.celulas[ vecinoX[1] ][ vecinoY[1] ].instrumento
        , todos.celulas[ vecinoX[2] ][ vecinoY[2] ].instrumento
        , todos.celulas[ vecinoX[3] ][ vecinoY[3] ].instrumento
        , todos.celulas[ vecinoX[4] ][ vecinoY[4] ].instrumento
        , todos.celulas[ vecinoX[5] ][ vecinoY[5] ].instrumento
        , todos.celulas[ vecinoX[6] ][ vecinoY[6] ].instrumento
        , todos.celulas[ vecinoX[7] ][ vecinoY[7] ].instrumento
        , x
        , y
    );

}

```

**Algoritmo 17: función actualizarCelula.**

```
void actualizarCelula( Instrumento este , Instrumento n , Instrumento ne , Instrumento e , Instrumento se ,
Instrumento s , Instrumento so , Instrumento o , Instrumento no , float x , float y ){
}
}
```

Como vimos en el algoritmo 15, la función actualizar del autómata, ejecuta el actualizar de la célula y a su vez envía los punteros a cada una de ella. Aquí también se le asigna un valor booleano a una variable llamada menuHabilitado, de vital importancia para el funcionamiento integral de la interfaz. Esta es la variable utilizada para que no se solapen las acciones de todas las células y que solo se ejecute un menú por vez, como hicimos referencia anteriormente. Permite que todas las células sepan cual es el menú que se está ejecutado. Cuando un menú es abierto, las demás células quedan inhabilitadas.

A su vez, esto es posible por dos variables booleanas declaradas dentro del instrumento: abrioMenu y cerroMenu. Cuando un menú es abierto, la variable menuHabilitado es definida en false, y si se cerró el menú es definida en true. Es decir, cuando cerroMenu es true, vuelve a habilitar todos los menús, hasta q alguno sea nuevamente abierto.

**Algoritmo 18: función actualizar de la clase Celula.**

```
void actualizar( Punteros punteros , boolean menuHabilitado ){
    instrumento.actualizar( punteros , menuHabilitado );
}
}
```

También existe una función actualizar perteneciente a la clase Instrumento, que recibe de la célula, que a su vez lo recibe del autómata, el objeto puntero, para habilitar o no el menú.

Con un ciclo for, recorre todos los punteros, y verifica si el menú no fue abierto y si está habilitado, además calcula si la posición del puntero del mouse coincide con la del menú, y por ultimo verifica si el botón de abrir/cerrar menu fue presionado. Si cumple con esas condiciones, en ese momento abrioMenu se cambia su valor a true. Lo realiza solo en ese fotograma, capta el momento en que se abre el menú.

De lo contrario, si no se cumplen las condiciones, significa que el menú está abierto, por lo que ejecuta la función ejecutarMenu.

**Algoritmo 19: función actualizar de la clase instrumento.**

```

void actualizar( Punteros punteros , boolean menuHabilitado){
    abrioMenu = false;
    cerroMenu = false;
    celula.actualizar();
    parametro.actualizar();
    for( int i=0 ; i<punteros.cantPunteros ; i++ ){
        Puntero este = punteros.punteroPorOrden( i );
        if( !menuAbierto ){
            if( menuHabilitado ){
                if( este.down && este.x > left && este.x<left+ancho && este.y > top && este.y<top+alto ){
                    menuAbierto = true;
                    abrioMenu = true;
                }
            }
        }
        else{
            ejecutarMenu( este );
        }
    }
}

```

**EjecutarMenu** le pasa el puntero correspondiente a cada uno de los botones o controles del menú, y a continuación a las barras de opciones. Básicamente su función es controlar si se está presionando o no con el mouse en el lugar indicado, ya sea para abrir un menú o si estuviera abierto para controlar la barra.

**Algoritmo 20: comportamiento EjecutarMenu de la clase Instrumento**

```

void ejecutarMenu( Puntero este ){

    encendido.actualizar( este );
    if( encendido.down ){
        cerrarMenu();
    }
    anterior.actualizar( este );
    if( anterior.down ){
        paginaMenu = ( (paginaMenu-1) < 0 ? 7 : (paginaMenu-1) );
    }
}

```

```

siguiente.actualizar( este );
if( siguiente.down ){
    paginaMenu = ( paginaMenu + 1 ) % 8;
}

```

Como veíamos anteriormente al comienzo del escrito, dentro del comportamiento miDraw del cuerpo principal del programa, se realiza la impresión del autómata (ver algoritmo 5) ejecutando el comportamiento imprimir de la clase AutomataCelular. Este comportamiento ejecuta la impresión de todas las células, de las celdas propiamente dichas (ver figura 1), posteriormente se imprimen los menús, solo un menú por vez en realidad. Este orden es para que los menús queden por encima de las celdas de las células.

**Algoritmo 21: comportamiento imprimir de la clase AutomataCelular.**

```

void imprimir(){
    for( int j=0 ; j<fil ; j++){
        for( int i=0 ; i<col ; i++){
            celulas[i][j].imprimir();
        }
    }
    for( int j=0 ; j<fil ; j++){
        for( int i=0 ; i<col ; i++){
            celulas[i][j].instrumento.imprimirMenu();
        }
    }
}

```

La impresión del menú, se realiza a través del comportamiento imprimirMenu de la clase Instrumento. El comportamiento controla que el menú este abierto, de ser así ejecuta una serie de opciones de base, y de acuerdo al valor de paginaMenu ejecuta la impresión de los parámetros en grupos de 5, según corresponda (ver figura 2).

**Algoritmo 22: comportamiento imprimir de la clase Instrumento.**

```
void imprimirMenu(){
if( menuAbierto ){
  rectMode( CORNER );
  textFont( fuente , m );

  fill( fondoMenu );
  rect( leftMenu , topMenu , m*20 , m*16 );

  fill( letrasMenu );
  text( “ Instrumento “+id , leftMenu+m*4 , topMenu+m*3 );

  encendido.imprimir();
  anterior.imprimir();
  siguiente.imprimir();

  if( paginaMenu == 0 ){
    I_on.imprimir();
    r_silencio.imprimir( );
    rit_tempo.imprimir( );
    rit_figura.imprimir( );
    rit_rubato.imprimir( );
  }
  else if( paginaMenu == 1 ){
    rit_chord_min.imprimir( );
    rit_chord_max.imprimir( );
    pcs.imprimir( );
    textura.imprimir( );
    acorde_time.imprimir( );
  }
  else if( paginaMenu == 2 ){
    r_permutacion.imprimir( );
    r_acorde_dir.imprimir( );
    r_arpeggio_princ.imprimir( );
    r_trino_rit_start.imprimir( );
    r_trino_rit_end.imprimir( );
  }
}
```

```

else if( paginaMenu == 3 ){
    r_trino_din_start.imprimir( );
    r_trino_din_end.imprimir( );
    r_ricochet_rit_start.imprimir( );
    r_ricochet_rit_end.imprimir( );
    r_ricochet_din_start.imprimir( );
}
else if( paginaMenu == 4 ){
    r_ricochet_din_end.imprimir( );
    r_transp.imprimir( );
    r_transp_start.imprimir( );
    r_registro.imprimir( );
}
else if( paginaMenu == 5 ){
    r_registro_start.imprimir( );
    r_regMin.imprimir( );
    r_dir.imprimir( );
    r_regMax.imprimir( );
    r_vel_start.imprimir( );
}
else if( paginaMenu == 6 ){
    r_vel_end.imprimir( );
    r_vel_time.imprimir( );
    r_acent_vel_increment .imprimir( );
    r_acent.imprimir( );
    r_acent_tipo.imprimir( );
}
else if( paginaMenu == 7 ){
    r_dur.imprimir( );
    r_dur_factor.imprimir( );
    r_pedal.imprimir( );
}

}
}

```

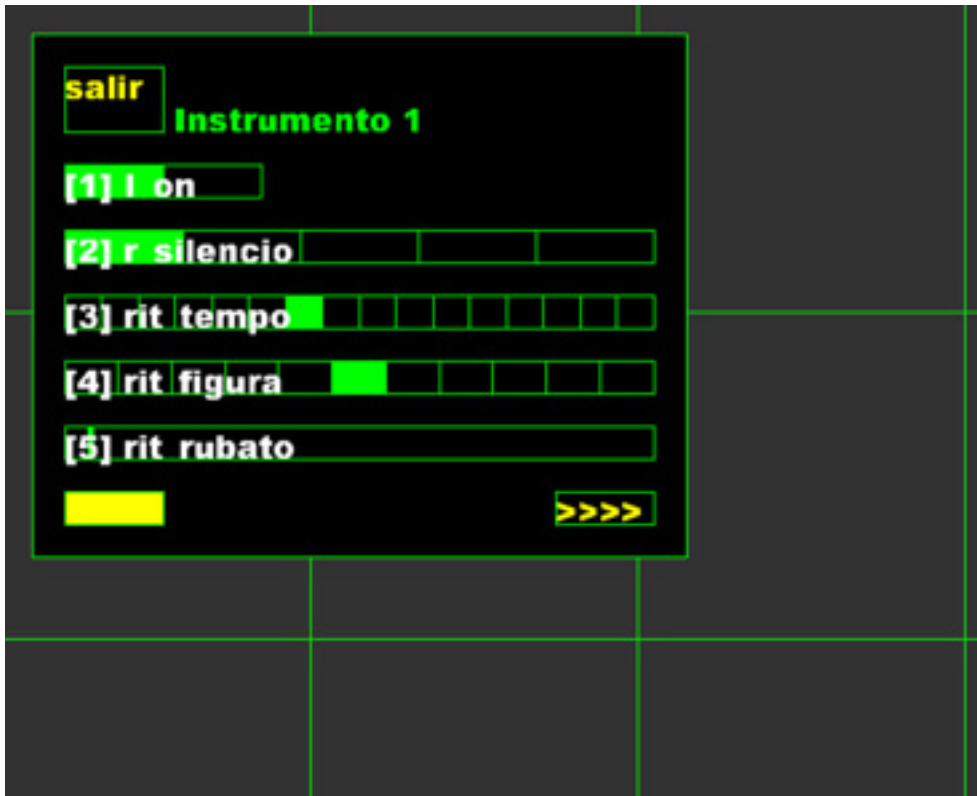


Figura 10 : impresión de una página de un menú

## 6. La interfaz física

La interfaz física utilizada es una pantalla de acrílico sensible al tacto, la cual funciona con una cámara web que capta la silueta de la punta de los dedos sobre la superficie.

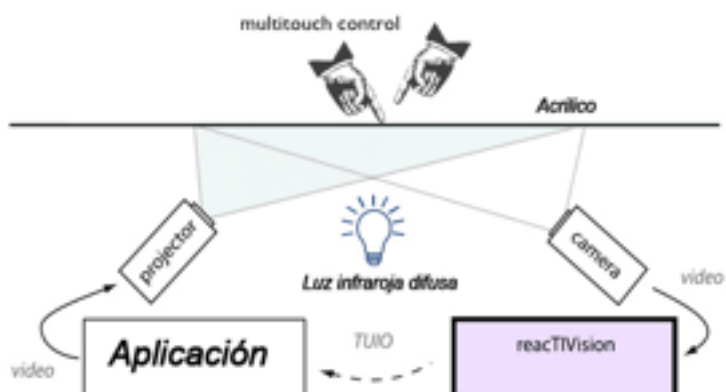


Figura 11 : esquema de la interfaz física

## 6.1. Vinculación Cliente Servidor entre el núcleo del programa e interfaz

Para la implementación de la interfaz virtual se utilizó la plataforma de Reactivision<sup>7</sup>, que actúa como servidor. Esta aplicación toma las imágenes capturadas por la cámara, las analiza determinando la cantidad y posición de puntos de presión sobre la superficie (dedos) y envía esa información a Processing, que funciona como cliente.

Reactivision funciona como un programa independiente y envía mensajes con los datos de captación a través de un puerto específico al autómata celular de Processing. Para el envío de estos datos se implementa el protocolo TUIO, especialmente diseñado para la transmisión de información relativa a objetos tangibles.

## 7. Un caso de aplicación

El siguiente es un ejemplo concreto de un autómata celular de dos dimensiones, con 64 células dispuestas en una grilla de 8 por 8, donde sus fronteras son periódicas (conforma un toroide), y su evolución se produce en pasos discretos. Cada célula está concebida como un instrumento complejo con las siguientes características:

- Cada columna está asociada a un timbre instrumental diferente.
- Como consecuencia de lo anterior, las 8 células de cada columna se puede pensar como voces independientes del instrumento.
- El tempo es fijo y el valor del pulso es igual a 90, es decir 90 pulsos de negra por minuto.
- El conjunto de grados cromáticos elegidos es el 4-26, cuya forma prima es 0 3 5 8.
- El registro de cada célula es fijo, y está restringido a una octava.
- No hay alteración de la regularidad del pulso por acentuaciones dinámica.

Los parámetros elegidos para ser afectados por la evolución del autómata fueron:

- Valor de la figura
- Duración de los silencios
- Dinámica
- Duración de los sonidos

---

<sup>7</sup> Reactivision: Aplicación open source para el desarrollo de pantallas táctiles. <http://reactivision.sourceforge.net/>

Para hacer más perceptibles la modificación de estos parámetros, las células fueron agrupadas por filas y repartidas en diferentes registros. Para ello se siguió un criterio de analogía entre la ubicación espacial en la grilla y en el registro. Así, las células ubicadas en la primera fila se desenvuelven en el registro más agudo y los de la última en el más grave. En nuestra grilla de de 8 x 8 cada fila está asociada a una octava diferente, desde la 0 a la 8.



```
int ancho = 8;
int tesitura = 8;
int posY = (este.id - 1) / ancho;
int octava = tesitura - posY;
este.set_r_registro_start (octava);
```

En este autómata de 8 x 8, cada una de las 64 células es en realidad un “instrumento” autónomo con funciones musicales de alto nivel, lo que supone una polifonía de 64 voces independientes. Si decidimos que todas ellas suenen al mismo tiempo, y más aún todo el tiempo, implicaría el múltiples movimientos de parámetros en cada nuevo estado, lo que provocaría, como se dijo en un principio, la saturación de la percepción.

Entonces, una solución posible es asignar turnos a cada célula en los que estén habilitados para “tocar” y momentos en los que deben “esperar su turno” en silencio, más allá de que sus parámetros internos puedan seguir variando. Estos turnos son determinados por cuatro variables:

- **tiempoVivo:** es la antigüedad de vida de la célula, hace cuánto tiempo que está “sonando”
- **tiempoMuerto:** es el tiempo que estuvo muerta la célula.
- **cuentaRegresiva:** es una cuenta regresiva para volver a vivir.
- **tiempoVida:** es su tiempo de vida asignado.

El proceso sigue la siguiente lógica: se dispara una cuenta regresiva al azar, que al llegar a 0 habilita a la célula para sonar. Una vez que la célula “nace” se le asigna un tiempo límite de vida (tiempoVida), también al azar, y comienza a correr su antigüedad (tiempoVivo). Una vez cumplido ese período, la célula deja de sonar y nuevamente empieza a correr la cuenta regresiva, luego de la cual, será el turno para la célula de volver a nacer.

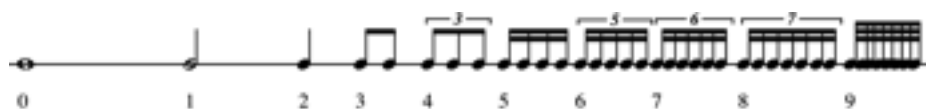
Como sabemos, en un autómata celular, el estado futuro de cada célula depende de su estado actual, así como del estado actual de sus vecinos. Esta particularidad es la que provoca que el autómata se desarrolle de manera orgánica. En nuestra aplicación, además de lde “vida”, solo aquellas células que no tengan vecinos vivos pueden nacer.

```
int vecinosVivos = e.get_I_on() + o.get_I_on() + n.get_I_on() + s.get_I_on() + se.get_I_on() + ne.get_I_on() + so.get_I_on() + no.get_I_on();
if ( vecinosVivos < 1)
```

La variable `e.get_I_on()` retorna el estado de vida de la célula ubicada al “este” o la derecha, siendo 1 si está sonando, y 0 si no. Lo mismo sucede con las demás células vecinas, ubicadas en todos los puntos cardinales, respecto del centro, oeste, norte, sur, sudeste, noreste, etc.

La figura es el parámetro que guía los intervalos de ataque entre las notas. No hay que confundir a la figura con la duración del sonido, se trata en realidad del intervalo de tiempo que separa dos entradas, o dos ataques. Por ejemplo la entrada de dos sonidos sucesivos puede estar separada por un segundo, pero al mismo tiempo, cada sonido puede durar medio segundo.

El valor de la figura va desde el tiempo de blanca (0), al de semifusas (9), pasando por valores regulares e irregulares intermedios, como se muestra en la imagen de abajo.



Los cuatro parámetros descritos al comienzo de esta sección están enlazados al valor de la figura, es decir que tanto el valor de la figura como los silencios, la dinámica y la duración de las notas dependen de la densidad cronométrica de sus vecinos. A diferencia de lo anterior, en este caso, solo se tienen en cuenta los vecinos más próximos que están ubicados a la izquierda (oeste) y derecha (este) de cada célula.

```
if( o.get_rit_figura() > este.get_rit_figura() || e.get_rit_figura() > este.get_rit_figura() )
```

Por defecto, todas las células comienzan con una figura de negra, esto provoca un primer movimiento: si los vecinos al este y oeste tienen la misma figura, como es el estado inicial, se asigna un nuevo valor al azar.

```
else{
    int nuevoFigura = int( random( 2 , 10 ));
    este.set_rit_figura( nuevoFigura );
}
```

En cambio, si la célula está ejecutando valores de corchea, pero sus vecinos al este y al oeste lo hacen en semicorcheas y tresillos de corcheas respectivamente, en su estado siguiente la célula incrementará su figura en una posición, es decir que ejecutará valores de tresillos de corchea.

Lo mismo sucederá con los silencios que serán más largos, y la duración de las notas, que serán más cortas, y su dinámica menor.

Más allá del valor estético de este ejemplo, es notable cómo surgen comportamientos de organización musical, en especial la alternancia de la aparición de las voces junto con la complementariedad rítmica entre ellas, como producto de la combinación de figuras regulares e irregulares de diferentes velocidades.

```

if (este.get_I_on () == 1){ //si esta vivo

    celula.tiempoVivo ++;
    if (celula.tiempoVivo > celula.tiempoVida){ //se paso su tiempo y muere
        este.set_I_on( 0 );
        celula.cuentaRegresiva = int( random( cuentaMin , cuentaMax ));
    }
}
else{//si esta muerto

    if( celula.cuentaRegresiva == 0 ){ //si le toca nacer

        int vecinosVivos = e.get_I_on () + o.get_I_on () + n.get_I_on () + s.get_I_on () + se.get_I_on () + ne.get_I_on () + so.get_I_on
() + no.get_I_on ();

        if ( vecinosVivos < 1){ // si no tiene vecinos vivos, nace.

            este.set_I_on( 1 );
            celula.cuentaRegresiva = -1;
            celula.tiempoVivo = 1;
            celula.tiempoVida = int ( random ( tiempoMinVida, tiempoMaxVida));
            if ( o.get_rit_figura() > este.get_rit_figura() || e.get_rit_figura() > este.get_rit_figura() ){
                int nuevoFigura = este.get_rit_figura();
                nuevoFigura ++;
                nuevoFigura = int( constrain( nuevoFigura , 2 , 10 ) );
                este.set_rit_figura( nuevoFigura );
                int nuevoSilencio = este.get_r_silencio();
                nuevoSilencio ++;
            }
        }
    }
}

```

```

nuevoSilencio = int( constrain( nuevoSilencio , 1 , 6 ) );
este.set_r_silencio( nuevoSilencio );
int nuevoDuracion = este.get_r_dur();
nuevoDuracion -= 5;
nuevoDuracion = int( constrain( nuevoDuracion , 1 , 40 ) );
este.set_r_dur( nuevoDuracion );
int nuevoDinamica = este.get_r_vel_start();
nuevoDinamica -= 10;
nuevoDinamica = int( constrain( nuevoDinamica , 60 , 120 ) );
este.set_r_vel_start( nuevoDinamica );
este.set_r_vel_end( nuevoDinamica );
}
else if( o.get_rit_figura() < este.get_rit_figura() || e.get_rit_figura() < este.get_rit_figura() ){
    int nuevoFigura = este.get_rit_figura();
    nuevoFigura --;
    nuevoFigura = int( constrain( nuevoFigura , 2 , 10 ) );
    este.set_rit_figura( nuevoFigura );
    int nuevoSilencio = este.get_r_silencio();
    nuevoSilencio --;
    nuevoSilencio = int( constrain( nuevoSilencio , 1 , 6 ) );
    este.set_r_silencio( nuevoSilencio );
    int nuevoDuracion = este.get_r_dur();
    nuevoDuracion += 5;
    nuevoDuracion = int( constrain( nuevoDuracion , 1 , 40 ) );
    este.set_r_dur( nuevoDuracion );
    int nuevoDinamica = este.get_r_vel_start();
    nuevoDinamica += 10;
    nuevoDinamica = int( constrain( nuevoDinamica , 60 , 120 ) );
    este.set_r_vel_start( nuevoDinamica );
    este.set_r_vel_end( nuevoDinamica );
}
else{
    int nuevoFigura = int( random( 2 , 10 ) );
    este.set_rit_figura( nuevoFigura );
}
}

if( o.get_r_silencio() > este.get_r_silencio() || e.get_r_silencio() > este.get_r_silencio() ){

```

```
else { // sino vuelve a probar mas adelante
    celula.cuentaRegresiva = int( random( cuentaMin , cuentaMax ));
}
}
else if(celula.cuentaRegresiva > 0){
    celula.cuentaRegresiva--;
}
}
}
```

## 8. Resultados

A partir de los temas tratados aquí, y en especial del análisis del caso concreto de aplicación del autómata musical, podemos arribar a una serie de conclusiones que dan cuenta de las posibilidades a futuro de estos desarrollos, así como de sus límites.

1. Si bien existe una gran cantidad de parámetros que pueden contribuir a la evolución musical, la representación gráfica de la interfaz fue dividida en dos partes, por un lado la visualización de la actividad del autómata, donde solo se reresetan aquellos parámetros que son más relevantes o que pueden asociarse más fácilmente con la percepción auditiva, y por otro lado la interfaz usuario que permite manipular todos los parámetros. Esta limitación se debe a que resulta muy complejo con este tipo de gráfica bidimensional, representar múltiples parámetros simultáneamente.
2. La ubicación horizontal de las células en diferentes registros, así como su distribución vertical en diferentes timbres hace más clara la audición de los comportamientos individuales de cada una de ellas. En el caso del registro, cada una de las ocho células puede tomar su altura dentro del ámbito de una octava completa, lo que nos da un registro total de 8 octavas. Sin embargo esto limita el número máximo de filas, ya que el ámbito que puede captar el oído humano es de aproximadamente 10 octavas<sup>1</sup>, sin embargo
3. Una de las características de los fenómenos emergentes es su impredecibilidad, por eso es sumamente difícil prever el comportamiento del autómata. Esta dificultad se vuelve relevante cuando se quiere generar una direccionalidad musical. Por ejemplo para crear una progresión de la velocidad de ejecución de las notas como un modo de construir un gesto de tensión y distensión por la aumento de la densidad crométrica.
4. La concepción de la célula como un “instrumento” complejo permite entrecruzar parámetros, es decir, vincular dos o más células musicales de forma no solamente horizontal (el mismo parámetro) sino diagonal, y lo que es más importante entre varios parámetros simultáneamente. Esto permite comportamientos análogos entre diferentes parámetros.

---

<sup>8</sup> El umbral de audición de la frecuencia en el ser humano tiene un rango aproximado que va des los 20Hz a los 20.000Hz, que pasados a octavas es igual a  $\log_2(20.000/20) = 9,965$  octavas.

Para poner un ejemplo, imaginemos que mientras los intervalos entre las alturas se hacen más pequeños, los intervalos de ataque y las duraciones de las notas se hacen más cortas. En este caso, el desarrollo rítmico es derivado de la estructuración de la altura.

5. Uno de los parámetros que no fue explotado en el ejemplo de aplicación es el de la altura, quizás el parámetro más importante en la música occidental hasta mediados del siglo XX. Se vuelve imprescindible entonces lograr una organización de la altura en sus relaciones verticales (armonía) como horizontales (contrapunto). Esto implica formalizar una serie de reglas de combinación de alturas que pueda ser conducida por la evolución del autómata. En nuestro caso, la elección se sustenta en la teoría de los conjuntos de grados cromáticos o pitch class sets.

## 9. Conclusión

A lo largo de esta investigación hemos comprobado que es posible establecer un grado de coherencia musical sustentándose en el desarrollo de un sistema complejo. Más importante aún, es que parece posible lograr la emergencia de estructuras musicales a partir de la auto-organización de un sistema autónomo.

Hemos visto que resulta poco fructífero asociar a una célula solamente con un sonido, y pretender entonces que el autómata se encargue de crear una composición musical. Esto se debe a la dificultad de equiparar un entidad restringida a un número finito de estados posibles, como la célula, a estructuras musicales complejas, con múltiples niveles de organización sintáctica. Para decirlo de otra manera, ¿cómo representar con un valor numérico un instante de música donde conviven simultáneamente relaciones armónicas, melódicas, rítmicas, texturales, dinámicas, tímbricas, etc., etc.? La verdadera dificultad se encuentra, entonces, en la elección de los estados en cada momento, y sobre todo, de las relaciones que se establezcan entre ellos.

Por esa razón nos pareció conveniente que cada célula constituya en realidad un conjunto complejo de unidades mínimas de sentido musical, y trasladar la problemática a la conducción orgánica de estas pequeñas estructuras a lo largo del tiempo.

Somos conscientes que este trabajo es solo un pequeño avance en este sentido, y que es necesario una exploración mucho más profunda para lograr estructuras musicales auto-organizadas que alcancen en toda su magnitud el nivel de una obra musical.

Matías Romero Costas - Emiliano Causa - Tarcisio Lucas Pirotta

Junio - 2009

## 10. Referencias bibliográficas

- [1] Fritjof Capra, “La Trama de la Vida”. Ed. Anagrama. Barcelona. 2006.
- [2] Stephen Wolfram, “A new kind o science”, Library of Congress Cataloging-in-Publication Data. Canada. 2002.
- [3] Stephen Wolfram: “A new kind o science: Online”, <http://www.wolframscience.com/nksonline/toc.html>, consultada 22/Dic/2007, actualmente en línea.
- [4] Stephen Wolfram: “Articles on Cellular Automata”, <http://www.stephenwolfram.com/publications/articles/ca/>, consultada 22/Dic/2007, actualmente en línea.
- [5] “Cellular Automata FAQ”, Editado por Tim Tyler. Originally edited by Howard Gutowitz, <http://cafaq.com/>, consultada 20/Dic/2007, actualmente en línea.
- [6] Stephen Wolfram , “Some Historical Notes”, <http://www.wolframscience.com/reference/notes/876b>, consultada 3/Ene/2008, actualmente en línea.
- [7] Juan Carlos López (2005), “A Language for the cellular Automata”, <http://cellularautomaton.blogspot.com/>, consultada 3/Ene/2008, actualmente en línea.
- [8] Miranda, E. R., “Composing Music with Computers”. Oxford (UK): Focal Press. 2001.
- [9] F. Richard Moore, “Elements of Computer Music”, PTR Prentice Hall Inc. Ney Jersey. 1990.
- [10] Charles Dodge y Thomas A. Jerse, “Computer Música”. Library of Congress. USA. 1997.
- [11] Allen Forte, “The structure of atonal music”. Yale University Press. 1973.
- [12] Pablo Cetta, “Principios de estructuración de la altura empleando conjuntos de grados cromáticos”. Cuaderno N° 5 del Instituto de Investigación Musicológica “Carlos Vega”. 2004.
- [13] Emiliano Causa, Matías Romero Costas, “Vinculación entre imagen y sonido en los sistemas interactivos y de vida artificial”, Presentado en la Revista Electrónica Cibertronic perteneciente a la UNTREF – Argentina. 2007.