Desarrollo de una Aplicación con Interfaces Tangibles

Autor: Emiliano Causa

emiliano.causa@gmail.com, www.emiliano-causa.com.ar

Palabras claves

Interfaces Tangibles, Reconocimiento de Patrones Bitonales, ReacTiVision, Processing

Resumen

Este trabajo explica la aplicación de herramientas de software específicas en la construcción de un prototipo de instalación interactiva con interfaces tangibles. El objetivo principal es mostrar la forma en que los diferentes softwares (los de reconocimiento de patrones y los de control de sonido y video en tiempo-real) pueden vincularse, a fin de configurar una unidad integral, en donde la interface, la imagen y el sonido se mueven en forma conjunta.

1. Introducción

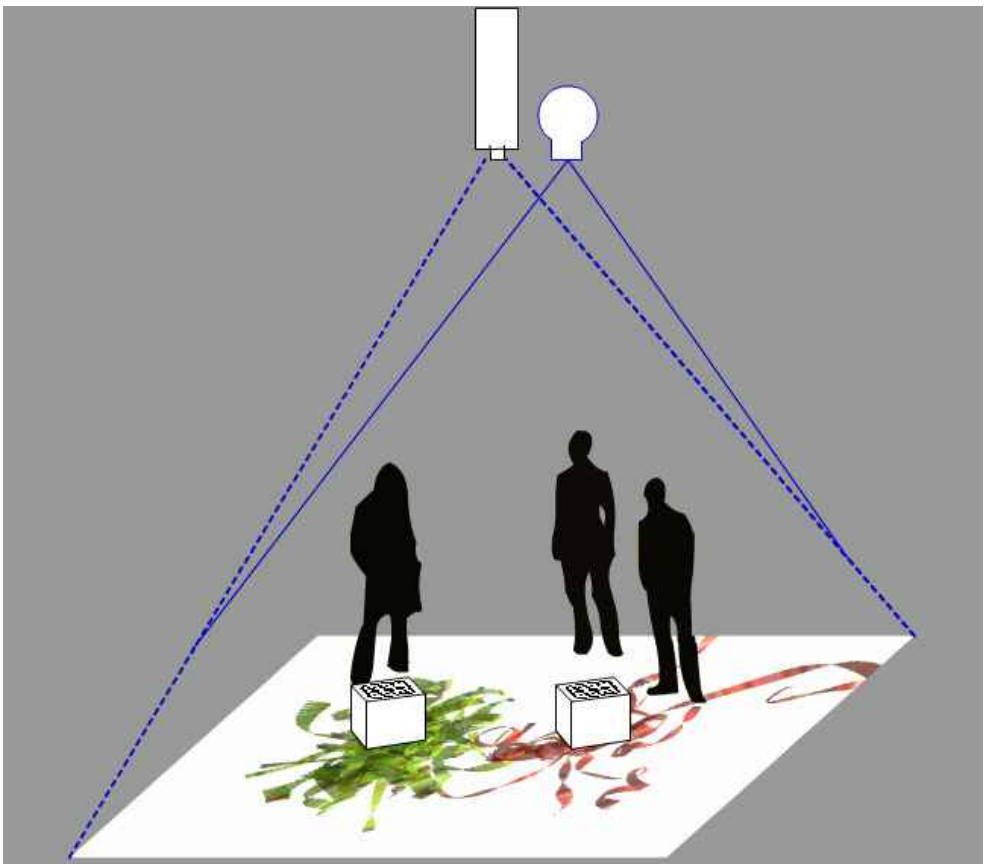

Figura 1: Esquema de la Instalación Interactiva

En este artículo mostraremos la construcción de una instalación interactiva con interfaces tangibles, aplicando herramientas de software destinadas al control de sonido y video en tiempo real, así como otras destinadas a la implementación de pantallas sensibles al tacto. Aquí vamos a mostrar la implementación de un prototipo de instalación con interfaces tangibles, que en este caso la interface estará constituida por unos cubos en el suelo que poseen patrones bitonales en sus caras. El público, al mover los cubos en el suelo, genera imágenes y sonidos que responden a la ubicación, presencia y rotación de estos. La idea consiste en que los cubos tengan impresos en sus caras dibujos con patrones bitonales. Una cámara en el techo transmitirá la imágenes de los patrones a un software con la capacidad de reconocerlos, el cual, en conjunto con otros, generará las imágenes (proyectadas por un cañon de video ubicado en el techo) y sonidos.

2. Distribución de tareas

En la construcción de este prototipo intervienen diferentes herramientas de software.

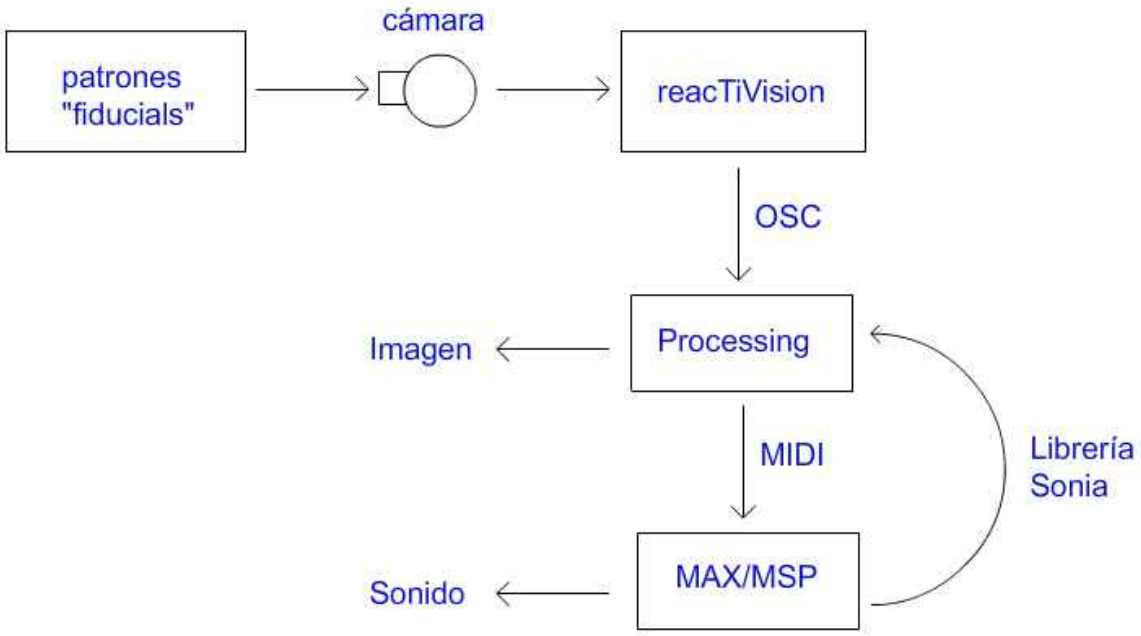

Figura 2: Esquema de flujo de información entre herramientas de software

La figura 2 es un esquema que muestra como se transmite el flujo de información entre los diferentes softwares que participan en la ejecución del prototipo. Para el reconocimiento de los patrones bitonales e implementación de la interface tangible usamos ReacTiVision. Este software transmite información de los patrones a Processing (vía el protocolo OSC), que es el encargado de generar la imagen en tiempo-real. A su vez, Processing transmite (vía protocolo MIDI) información a MAX/MSP, el cual se encarga de generar el sonido en tiempo-real. Este sonido vuelve, como un lazo de

retroalimentación a Processing, dado que Processing capta el sonido utilizando una librería llamada Sonia.

2.1. ReacTiVision y las Interfaces Tangibles

ReacTiVision (mtg.upf.es/reactable/) es una herramienta de software desarrollada por Sergi Jordà, Martin Kaltenbrunner, Günter Geiger y Marcos Alonso, quienes conforman el Grupo de Tecnología Musical dentro del Instituto Audiovisual en la Universidad Pompeu Fabra (Barcelona España). Esta tecnología permite reconocer patrones bitonales (llamados “fiducials”) impresos a piezas de interfaces tangibles que funcionan sobre una pantalla sensible al tacto. Esta interface tangible consiste en piezas de acrílico que se apoyan sobre una pantalla sensible, esta es capaz de reconocer las piezas, gracias a los patrones bitonales y generar respuestas audiovisuales en consecuencia. Los creadores, construyeron este software para desarrollar una pantalla sensible para interpretación musical en tiempo-real (un instrumento de improvisación de música electrónica) llamada ReacTable.

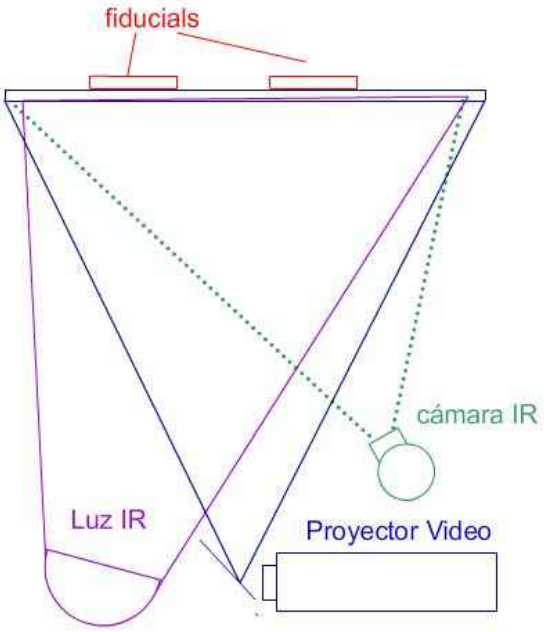

Figura 3: Esquema ReacTable

Como muestra la figura 3, ReacTiVision permite hacer el reconocimiento de patrones bitonales, a través de un sistema óptico, que en el caso de la ReacTable se implementa con luces y cámara infrarrojas. La pantalla es un acrílico con superficie esmerilada, las imágenes se retro-proyectan desde abajo usando un cañón de video, a su vez una luz infrarroja permite iluminar los patrones que serán captados por una cámara, también infrarroja. Dicha luz y cámara son infrarrojas para no interferir la luz del proyector de video (que pertenece al rango visible de la luz), y para que la cámara no vea a su vez las proyecciones.

Uno de los aspectos más interesantes de ReacTiVision es que está construido como un software independiente, que envía datos respecto de los parámetros de los “fiducials”: la ubicación,

identificación, rotación y otros; vía el protocolo OSC (Open Sound Control). Esto permite que cualquier otro software que reciba mensajes en OSC, pueda comunicarse con ReacTiVision e interpretar información respecto del estado de cada uno de los patrones bitonales ubicados sobre la pantalla. Debido a esto, existe en el sitio de ReacTiVision, ejemplos de conexión de este software con lenguajes como: C++, Java, C#, Processing, Pure Data, Max/MSP, Flash y otros.

2.2. Processing y la librería ReacTiVision

Processing (www.processing.org) es un lenguaje de programación diseño para artistas por Ben Fry y Casey Reas. Es un lenguaje “open source” desarrollado en Java de gran potencia y facilidad de aprendizaje. Una de las ventajas de Processing es el extenso desarrollo de librerías que extienden las posibilidades de conexión de este lenguaje con otros formatos, protocolos o lenguajes. Como hemos dicho en el apartado anterior, existe una librería que permite conectar, vía OSC, a Processing con ReacTiVision.

ReacTiVision tiene una aplicación ejecutable que se conecta a la cámara y reconoce los patrones (fiducials) que estén en la imagen, enviando por OSC los parámetros de cada patrón (en un protocolo que los autores llamaron TUIO). Esta librería implementa un conjunto de instrucciones que permiten leer dichos parámetros. Por ejemplo, la siguiente línea de código crea un objeto “cliente de protocolo TUIO” :

```
TuioClient client = new TuioClient(this);
```

Y las instrucciones que siguen, son funciones que se ejecutan frente a eventos provocados por los patrones:

```
void addTuioObject(int s_id, int f_id, float xpos, float ypos, float
angle) {}


void removeTuioObject(int s_id,int f_id ) {}


void updateTuioObject (int s_id, int f_id, float xpos, float ypos,
float angle, float xspeed, float yspeed, float rspeed, float maccel,
float raccel) {}


void addTuioCursor(int s_id, float xpos, float ypos) {}


void removeTuioCursor(int s_id) {}
```

```
void updateTuioCursor (int s_id, float xpos, float ypos, float
xspeed, float yspeed, float maccel) {}
```

Por ejemplo, TUIO distingue dos tipos de elementos: objetos y cursores. Los objetos son los patrones bitonales, mientras que los cursores son los dedos que se apoyan sobre la pantalla (dado que el sistema también es capaz de reconocer el tacto). Cada una de estas funciones, informan un evento de un patrón o de tacto:

addTuioObject: informa la aparición de un nuevo patrón sobre la pantalla.

removeTuioObject: informa que un patrón salió de la pantalla.

updateTuioObject: informa los cambio que sufre un patrón, ya sea de posición como de rotación.

addTuioCursor: informa la aparición de un dedo sobre la pantalla.

removeTuioCursor: informa que un dedo salió de la pantalla.

updateTuioCursor: informa los cambio que sufre un dedo, un cambio de posición.

2.3. Processing y MAX/MSP

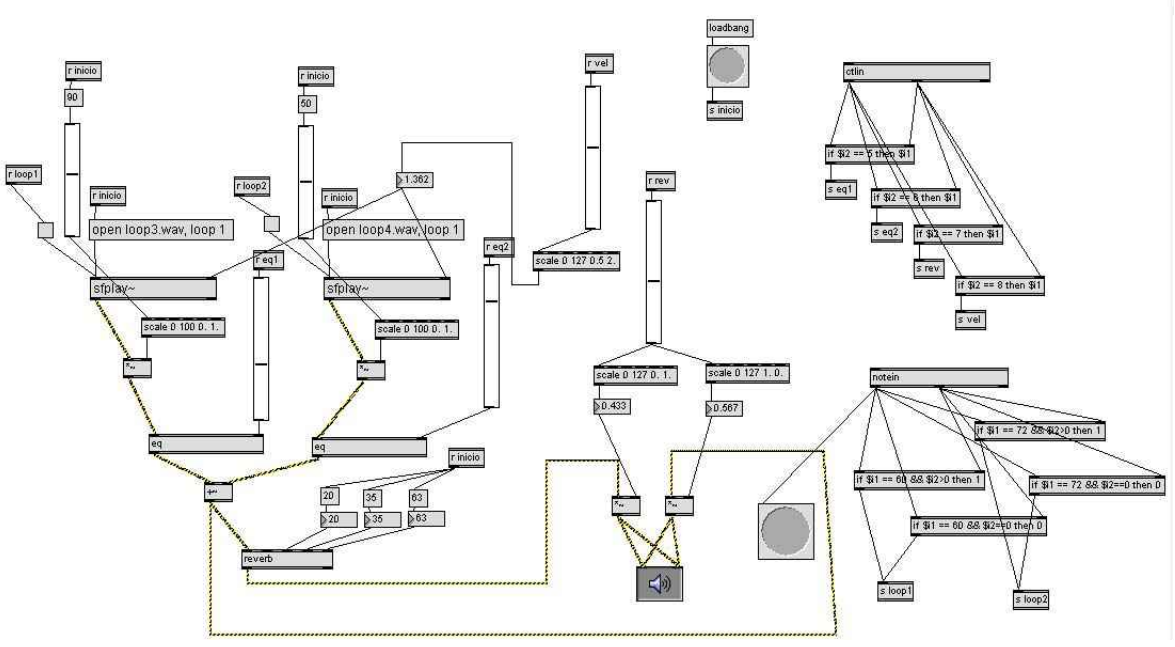

Figura 4: MAX/MSP

MAX/MSP (www.cycling74.com) es un lenguaje de control de sonido en tiempo real, posee un ambiente de programación visual que funciona mediante la conexión de objetos visuales. Su lógica de funcionamiento es a través del envío de mensajes de un objeto a otro y gran parte de sus funciones son para el tratamiento de mensajes MIDI, un protocolo que permite conectar instrumentos musicales

digitales. Debido a que Processing posee una librería llamada “proMidi” (www.texone.org/promidi/), esto permite conectar Processing con MAX/MSP usando el protocolo MIDI para comunicarlos.

2.4. La librería Sonia

La librería para Processing llamada Sonia (sonia.pitaru.com) permite hacer análisis espectral del sonido en tiempo-real, de esta manera es posible hacer que Processing “escuche” el sonido generado en MAX/MSP para que las imágenes respondan a su fluctuación.

```
import pitaru.sonia_v2_9.*;

void setup(){

      Sonia.start(this);

      LiveInput.start(16);

}

void draw(){

      for ( int i = 0; i < LiveInput.spectrum.length; i++){

            float a = LiveInput.spectrum[i];

      }

}
```

Como se puede ver en las líneas de código de arriba:

1- La primer línea permite importar la librería

2- Las líneas 3 y 4 inicializan la librería Sonia y abren una entrada de sonido para ser analizada usando un análisis espectral. El parámetro 16 pasado en la función **LiveInput.start(16)** declara que el análisis tendrá una precisión de 16 bandas.

3- El resultado de dicho análisis es volcado en un arreglo llamado **spectrum**. En las líneas 7 y 8 se recorre el arreglo **spectrum** con el fin de leer cada una de las bandas del análisis.

3. La implementación

Veremos ahora como desde Processing se genera la imagen en tiempo-real, en respuesta a los movimientos de los patrones bitonales. Posteriormente, abordaremos el envío de instrucciones MIDI que permiten que MAX/MSP genere el sonido.

3.1. Generación de la imagen

En Processing el flujo de ejecución del programa transita principalmente a través de dos funciones: **setup( )** y **draw( )**. A su vez existen un conjunto de eventos que ejecutan otras funciones, como **mousePressed( ), keyPressed( )**, etc. Como vimos en el apartado de “Processing y la librería ReacTiVision”, esta librería agrega funciones que responden a nuevos eventos relacionados con los cambios producidos en los “fiducials”.

La imagen del prototipo esta conformada por unas cintas que se despliegan en un espacio 3D virtual y que conforman una especie de racimo puesto en forma de flor.

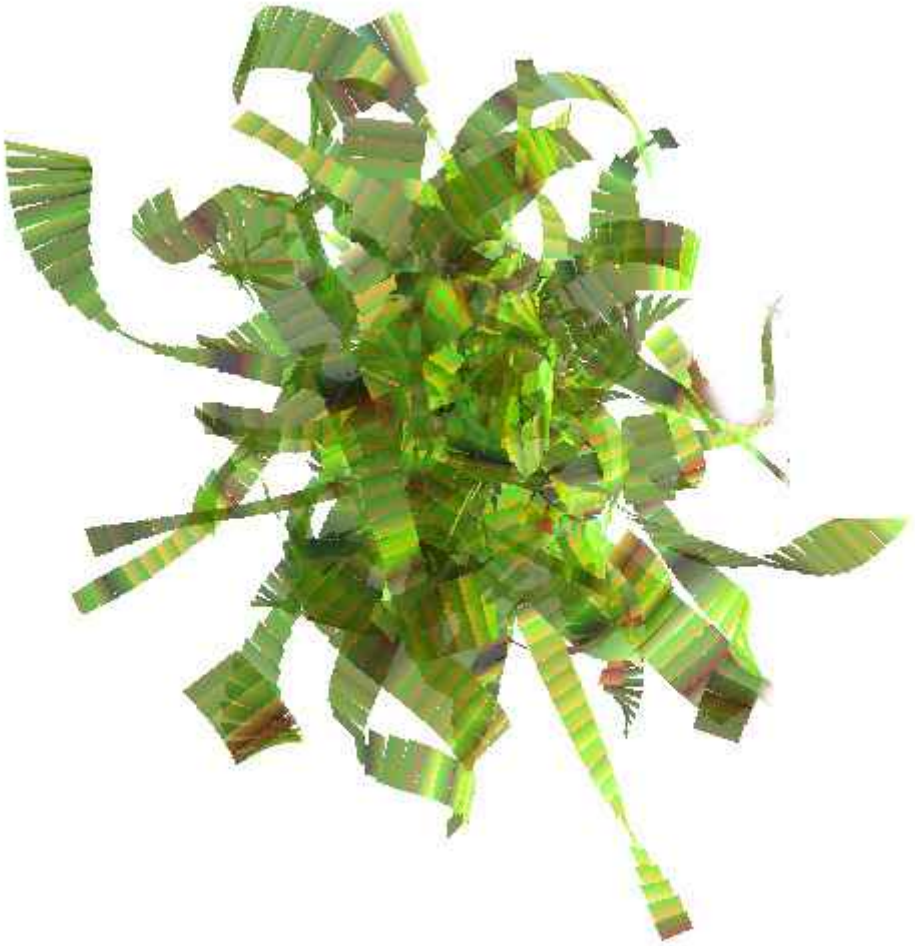

Figura 5: Racimo en forma de flor

Dichas formas están creadas con un objeto (una clase) llamada **Flor**:

```
Flor flor1;

void setup(){

      flor1 = new Flor( ... );

}
```

```
void draw(){
      flor1.actualizar();
      flor1.dibujar( ... );
}
```

(Los puntos suspensivos indican código que ha sido omitido.)

Este tipo de objeto responde a las siguientes instrucciones:

1- En la línea 1 se puede ver la declaración de un objeto tipo **Flor**, que en este caso llamamos **flor1**.

2- En la línea 3 está la ejecución del constructor de la clase, el cual inicializa el objeto.

3- En la línea 5 se ejecuta el comportamiento **actualizar()**, el cual, como bien dice su nombre, actualiza las variables internas del objeto, las cuales despliegan las cintas.

4- En la línea 6 esta el comportamiento que dibuja el objeto.

No profundizaremos en las instrucciones que implementan estas funciones, ya que las mismas no constituyen el objetivo central de este trabajo, sino una excusa que nos permite ejemplificar una implementación con interfaces tangibles. Pero volviendo a estos comportamientos y para analizar el modo en que se relacionan con los movimientos de los patrones en el escenario, podemos establecer el siguiente esquema:

1- Al iniciar la aplicación hay que declarar el objeto

2- El mismo debe inicializarse cada vez que el patrón entra en escena

3- Mientras el patrón está en escena, el objeto debe actualizarse y dibujarse

Siguiendo esta lógica, analizaremos las siguientes líneas de código:

```
import tuio.*;
Flor flor1 , flor2; //linea 2
TuioClient tuioClient; //linea 3
...
void setup(){
  ...
  tuioClient  = new TuioClient(this); //linea 7
  ...
}
void draw(){
  ...
```

```
  TuioObject[] tuioObjectList =tuioClient.getTuioObjects();//linea 12

  for (int i=0;i<tuioObjectList.length;i++) { //linea 14
    TuioObject tobj = tuioObjectList[i]; //linea 15

    if( tobj.getFiducialID() == patron1 ){ //linea 17
      if( flor1 != null ){ //linea 18
        flor1.actualizar();
        float x = tobj.getScreenX(width); //linea 20
        float y = tobj.getScreenY(height); //linea 21
        flor1.dibujar(...,x,y,tobj.getAngle() , ... ); //linea 22
      }
    }
    else if( tobj.getFiducialID() == patron2 ){
      if( flor2 != null ){
        flor2.actualizar();
        float x = tobj.getScreenX(width);
        float y = tobj.getScreenY(height);
        flor2.dibujar( ... , x , y , tobj.getAngle() , ... );
      }
    }
  }
  ...
}
void addTuioObject(TuioObject tobj) { //linea 36
  ...
  if( tobj.getFiducialID() == patron1 ){ //linea 38
    flor1 = new Flor( ... ); //linea 39
  }
  else if( tobj.getFiducialID() == patron2 ){
```

```
      flor2 = new Flor( ... );
    }
    ...
  }
```

En la línea 2 se declararon dos objetos de tipo **Flor**, llamados **flor1** y **flor2**. A su vez en la línea 3 se declara el objeto **TuioClient** que es el encargado de conectar a Processing con ReacTiVision.

La inicialización de los objetos se realiza en la función que se dispara ante el evento de aparición de un nuevo patrón en escena. La función, que está en la línea 36, pasa como parámetro un objeto de tipo **TuioObject** llamado **tobj** Dado que esta función se ejecuta frente a la aparición de cualquier patrón, es necesario identificar cuál es el patrón que disparó el evento. En la línea 38 se ejecuta el comportamiento **getFiducialID()**, el cual devuelve el número de identificación del patrón (ReacTiVision viene con un archivo formato PDF que tiene cientos de patrones para ser impreso, cada uno con su correspondiente número de identificación). Si el número de identificación coincide con el buscado, entonces en la siguiente línea se inicializa el objeto de tipo **Flor**.

La función **draw()** se ejecuta en “loop” y es la encargada de ejecutar el flujo de refresco de la pantalla de la aplicación (es decir de dibujar la pantalla). Es aquí en donde hay que actualizar y dibujar los objetos **Flor**. Pero esta acción debe estar condicionada por la presencia del patrón correspondiente, dado que ante la ausencia no deben realizarse. Por eso, en la línea 12 se carga en un arreglo de tipo **TuioObject** la lista de objetos TUIO presentes en escena. Luego, en la línea 14, se recorre con un ciclo “for” a dicho arreglo, tomando en la línea 15, cada uno de los objetos y cargándolo en un objeto llamado **tobj**. En la línea 17, nuevamente se verifica si la identificación del patrón coincide con el buscado, en cuyo caso se actualiza y dibuja el objeto. Previamente, en la línea 18, se revisa si el objeto **Flor** no es nulo (**null**) esto podría suceder si el objeto no fue debidamente inicializado (ya sea por que no se disparó el evento **addTuioObject**, o por que la identificación fue erronea).

En los objetos de tipo **TuioObject** se encuentran almacenados los parámetros de los patrones: la ubicación (mediante los parámetros **getScreenX( )** y **getScreenY( )**) y el ángulo de rotación con **getAngle( ).**

3.2. Envío de los mensajes MIDI

De la misma manera en que la imagen aparece, se modifica y desaparece en función del comportamiento de los patrones, asociando una imagen a cada patrón, existe un sonido asociado a cada uno de los patrones. La ejecución al “unísono” de la imagen y el sonido frente al comportamiento del patrón, une a estos fenómenos en un evento integral. La ejecución del sonido, como ya dijimos, la realiza MAX/MSP, y lo hace respondiendo a los mensajes MIDI que le envía Processing. Cuando MAX/MSP recibe un mensaje de **note on**, se enciende el sonido asociado al número de nota en cuestión, mientras que el mismo se detiene cuando recibe un mensaje **note off**. A

su vez, para ir variando características del sonido, Processing envía a MAX/MSP mensajes de controlador. Tanto las imágenes así como el sonido varían cuando los patrones rotan en su posición.

En el código presentado a continuación, veremos como son ejecutados los mensajes MIDI. Es importante destacar que ni en este, ni en el anterior código fuente, se encuentra el programa completo. Ambos son recortes del programa definitivo que buscan hacer foco sobre ciertas partes del mismo:

```
import promidi.*; //linea 1
MidiIO midiIO;
MidiOut midiOut;
...
void setup(){
  ...
  midiIO = MidiIO.getInstance(this);//linea 7
  midiIO.printDevices();
  midiOut = midiIO.getMidiOut(0,2);
  ...
}
void addTuioObject(TuioObject tobj) { //linea 12
  if( tobj.getFiducialID() == patron1 ){
    Note note = new Note(60,127,2000000);
    midiOut.sendNote(note);
  }
  else if( tobj.getFiducialID() == patron2 ){
    Note note = new Note(72,127,2000000);
    midiOut.sendNote(note);
  }
}
void removeTuioObject(TuioObject tobj) {//linea 22
  if( tobj.getFiducialID() == patron1 ){
    Note note = new Note(60,0,200000);
    midiOut.sendNote(note);
```

```
  }
  else if( tobj.getFiducialID() == patron2 ){
    Note note = new Note(72,0,200000);
    midiOut.sendNote(note);
  }
}
void updateTuioObject (TuioObject tobj) {//linea 32
  if( tobj.getFiducialID() == patron1 ){
    int valor = int( map( tobj.getAngle() , 0 , TWO_PI , 0 , 127  ));
    midiOut.sendController(new Controller(5,valor));
  }
  else if( tobj.getFiducialID() == patron2 ){
    int valor = int( map( tobj.getAngle() , 0 , TWO_PI , 0 , 127  ));
    midiOut.sendController(new Controller(6,valor));
  }
}
```

Los mensajes MIDI son enviados cuando acontecen eventos de los patrones. En las líneas 1, 2 y 3 se invoca la librería y declaran los objetos para el envío de MIDI, los que son inicializados en las líneas 7, 8 y 9. En la línea 9 se declara el número de puerto y canal MIDI que se utilizará. De la línea 12 a la 21 se puede ver como se disparan los mensajes **note on** cuando aparece un nuevo patrón. Como ya vimos en los ejemplos anteriores, dentro de la función que dispara el evento se ejecuta un condicional que verifica la identificación del patrón que lo originó. De la línea 22 a la 31 se ve la situación opuesta, cuando se quita un patrón de la escena, se ejecuta el mensaje **note off** del sonido correspondiente.

De la línea 32 hasta el final, se envían mensajes de controlador como respuesta al cambio de posición en los patrones. Los mensajes de controlador son utilizados para enviar una valor gradual que varía de 0 a 127 en función del ángulo de rotación del patrón. La función **map( )** se encarga de reescalar el valor del controlador de 0 a 127, cuando el ángulo varía de 0 a 2 *PI*.

3.3. El control de sonido desde MAX/MSP

Como hemos dicho en el apartado “Processing y MAX/MSP”, este lenguaje funciona mediante el envío de mensajes entre objetos. La información que envía Processing vía MIDI, es recibida por dos objetos: **notein** y **ctlin**:

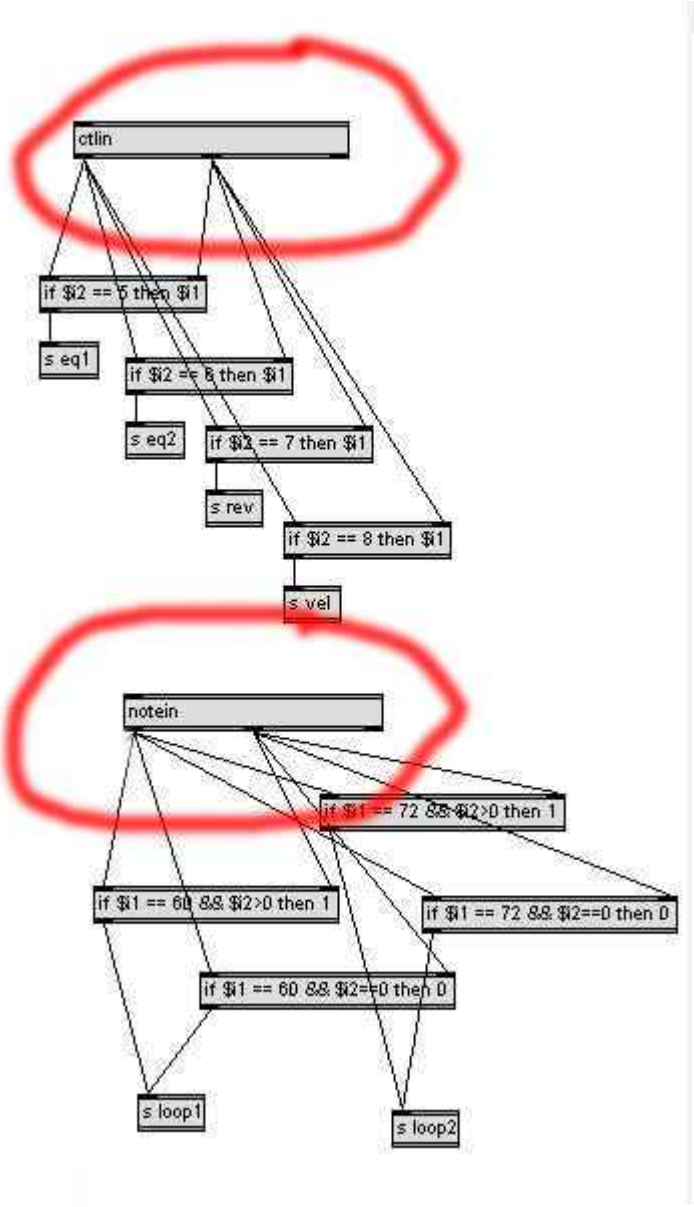

Figura 6: Detalle del programa de MAX/MSP

Como se ve en la figura 6, estos objetos reciben mensajes MIDI, los cuales son filtrados por los condicionales (**if…then**). Por ejemplo, si los mensajes de controlador corresponden a un número de controlador 5, entonces el valor del controlador es enviado a un objeto **send** (el que está etiquetada "**s eq1"**) que dirige el mensaje a un ecualizador, si es controlador 6 el mensaje se dirigirá a otro ecualizador, con el controlador 7 va a una reverberancia y con el controlador 8 a un control de velocidad.

El objeto **notein** se conecta a otra serie de objetos condiciones que filtran 4 tipos de mensajes: 1) nota número 60 con velocity mayor que cero, el cual enciende el loop 1, 2) nota 60 con velocity igual a cero, lo cual apaga el **loop 1**, y lo mismo para la nota 72, que enciende y apaga el **loop 2**.

Los mensajes MIDI pueden llegar de Processing a MAX/MSP gracias a un puerto MIDI virtual, en este caso usamos el software MIDIYOKE (www.midiox.com), el cual simula ser un dispositivo de hardware

(una interface midi) y permite que las aplicaciones lo seleccionen como un puerto MIDI para comunicarse entre si.

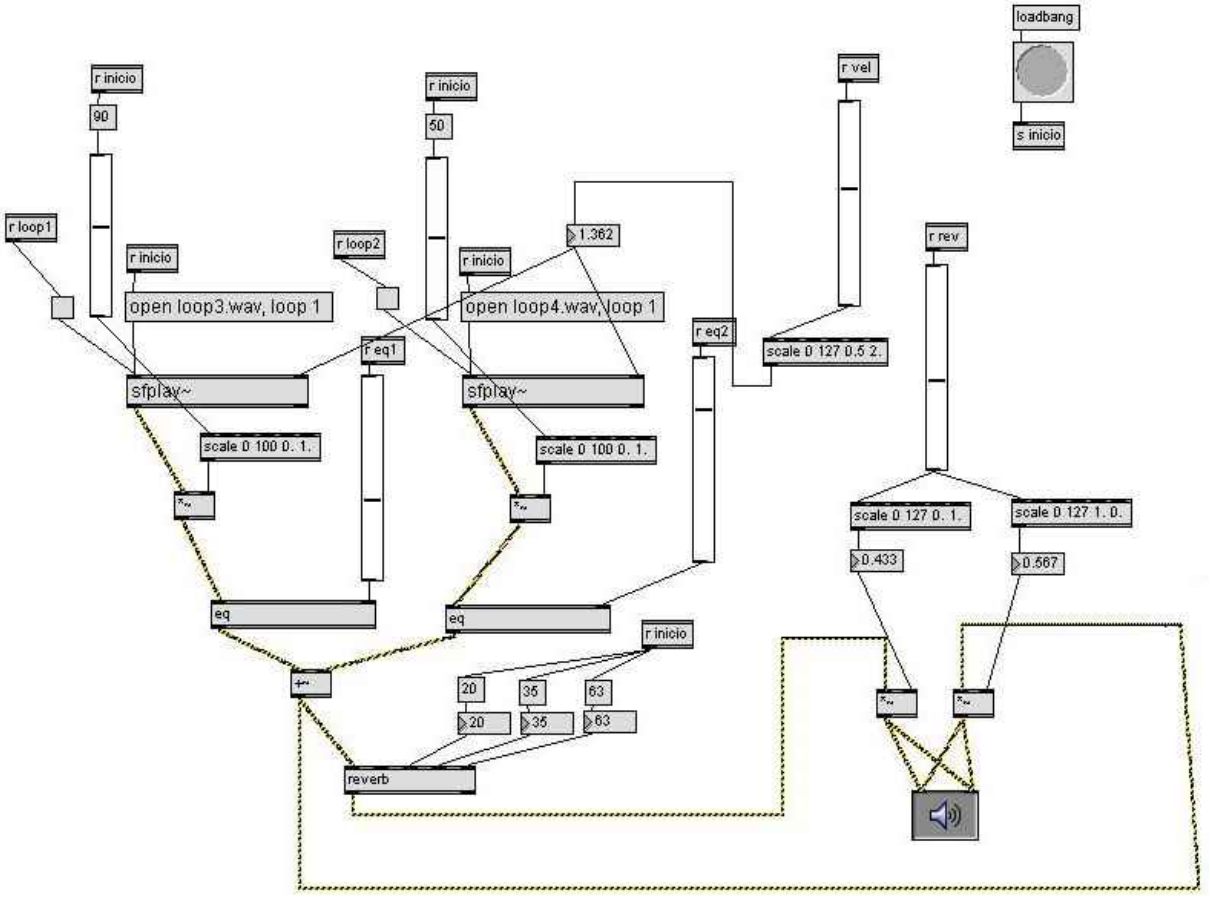

Figura 7: Detalle del programa de MAX/MSP

La reproducción del sonido está a cargo de dos objetos **sfplay** los cuales reciben la instrucción de encendido y apagado de los objeto **receive** (etiquetados "**r loop1"** y "**r loop2"**). A su vez, estos objetos transmiten el sonido a sendos objetos ecualizadores (etiquetados **"eq"**). Como última etapa el sonido para por una reverberancia (etiquetada **reverb**) y finalmente a la salida de sonido (con forma de parlante).

3.4. Retroalimentación del sonido

Por último, una vez que MAX/MSP emite el sonido como respuesta a los mensajes MIDI que envía Processing, la librería Sonia permite que este responda modificando la imagen en función de las fluctuaciones sonoras.

```
import pitaru.sonia_v2_9.*;

void setup(){

  ...
```

```
  Sonia.start(this); //linea 4
  LiveInput.start(16); //linea 5
  ...
}
void draw(){
  ...
  float graves = 0;//linea 11
  float agudos = 0;
  LiveInput.getSpectrum();

  for ( int i = 0; i < LiveInput.spectrum.length; i++){ //linea 15
    if( i< 2 ){ //linea 16
      graves += LiveInput.spectrum[i]; //linea 17
    }
    else if(i>=3 && i<6){ //linea 19
      agudos +=  LiveInput.spectrum[i]; //linea 20
    }
    graves = constrain( graves , 0 , 8000 );//linea 22
    agudos = constrain( agudos , 0 , 5000 );
  }
   modifAlfa = map( agudos , 0 , 8000 , 0 , amplitud ) * 0.2 +
modifAlfa *0.8;//linea 25
   modifBeta = map( graves , 0 , 5000 , 0 , amplitud ) * 0.2 +
modifBeta *0.8;

  flor1.dibujar( ... , modifAlfa , modifBeta , ... ); // linea 28
  ...
}
```

La librería Sonia permite realizar un análisis espectral. Como se puede ver en el código expuesto arriba, la línea 5 permite definir la cantidad de banda del análisis espectral en 16. El **ciclo for** de la linea 15 permite recorrer el espectro, sumarizando en la variable llamada **graves** las dos primeras componentes y en la variable llamada **agudos** las siguientes bandas, esta distribución de

componentes entre los graves y los agudos se debe conformación logarítmica de la gráfica espectral (en donde los graves son las primeras bandas y los medios y agudos ocupan cada vez mayor espectro). Los valores de estas variables son transformados (en las líneas 22 a la 26) en las variables **modifAlfa** y **modiBeta**, las cuales son enviadas como parámetros al comportamiento que dibuja la flor, esto permite que el sonido modifique el ángulo (en 3D) con el que se muestra la flor.

4. Conclusión

El prototipo recién descrito muestra como es posible vincular varias herramientas de software para la creación de una aplicación de interfaces tangibles. Es importante destacar que, a excepción de MAX/MSP, todo el software utilizado es open-source. MAX/MSP, sin embargo, puede ser fácilmente reemplazado por su versión “open source”, llamada Pure Data. La posibilidad de trabajar con software netamente open-source, habla del bajo costo que implica el desarrollo de este tipo de interfaces, lo que contrasta con el alto nivel de innovación y la potencialidad que la misma muestra.

5. Referencias bibliográficas

[1] www.processing.org
[2] mtg.upf.es/reactable/
[3] www.cycling74.com
[4] www.midiox.com
[5] sonia.pitaru.com

Emiliano Causa