“Ugarit” una interfaz gestual para la escritura musical analógica

Autores: Emiliano Causa, Matías Romero Costas, Sebastián G. Botasi, Jose Rafael Subia Valdez, Tarcisio Pirotta

Proyecto de investigación “Diseño y desarrollo de aplicaciones e interfaces de Realidad Aumentada destinadas a síntesis y procesamiento de audio digital.” – Director Carmelo Saitta, Codirector Pablo Cetta –Área de Artes Multimediales del Instituto Universitario Nacional de Arte – Buenos Aires (Argentina) – Yatay 843 Ciudad Autónoma de Buenos Aires Tel. 54-11-4862-8209

Palabras claves

Interfaces tangibles, Pantallas Sensible al Tacto, Aplicaciones Musicales, Software Open Source

Resumen

El presente documento sintetiza el progreso obtenido en el programa de investigación PICTO – Arte dirigido por Carmelo Saitta y el Dr. Pablo Cetta, en el cual se desarrollo una pantalla sensible al tacto, conocida como “Ugarit”, aplicada a un editor gestual de música, dicha interface es el trabajo conjunto de varias áreas de investigación. En este informe, se describen todos los procesos realizados en la elaboración del prototipo presentado, desde las investigaciones realizadas para el correcto sensado de la información ingresada realizado con ReacTiVision y Processing, hasta el análisis y re-mapeo de las alturas musicales realizado con Pure Data. “Ugarit” fue exhibida oficialmente el 23 de septiembre de 2011 en la Primera Jornada de Investigación en/de/sobre Artes, organizada por la secretaria de investigación y posgrado del IUNA.

1. Introducción

La pantalla sensible al tacto con aplicación musical desarrollada en 2011 posee un sistema de sensado de imagen que permite la escritura de gestos musicales. En la elaboración de dicha superficie, se implementaron programas de computación en el lenguaje de programación conocido como Processing. Dicho lenguaje permitió la creación de sistemas para la implementación en el proyecto.

La mesa táctil además fue armada con tecnología de bajo o adaptada con carácter “DIY” (Do It Yourself –hazlo tu mismo-). El ensamblado de la mesa táctil fue hecha por los propios miembros del equipo de investigadores. Para lograrlo se buscó y diseñó la mejor configuración posible de materiales. Dicha configuración fue documentada detalladamente por parte del equipo para poder producir un instrumento tecnológico de calidad.

La superficie interactiva conocida como “Ugarit” además posee sistemas complejos de estructuración de altura musical. Este sistema de ordenamiento de alturas musicales se desarrolló en el entorno de programación multimedial conocido como Pure Data. Dicho entorno cuenta además con la inclusión de una librería especializada en el tema desarrollada en uno de los proyectos de investigación del IUNA y permite un acercamiento novedoso hacia el entendimiento y la practica de la música occidental moderna contemporánea del siglo XX y XXI.

“Ugarit” recibe su nombre de la primera tableta con inscripciones musicales encontrada en la ciudad de medio oriente bautizada con el mismo nombre. El Himno de Ugarit es la obra musical escrita mas antigua del mundo. La superficie táctil desarrollada en este proyecto plantea un visión original y fresca al uso de tecnología moderna de este tipo. El proyecto mantiene conceptos musicales arraigados a la tradición pero además plantea una modernización al incluir teorías contemporáneas de música a su funcionamiento.

1.1. Esquema general del sistema

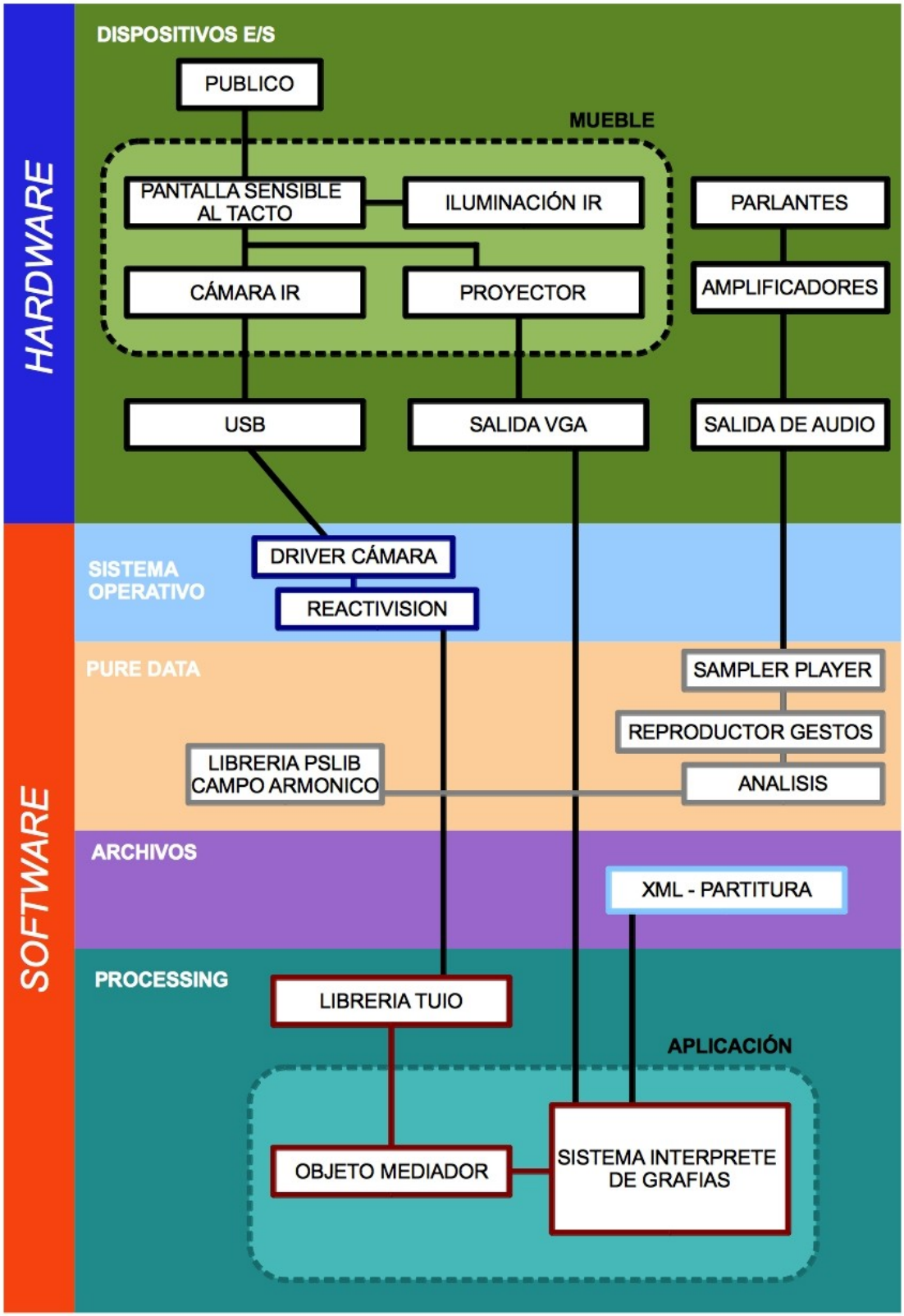

Figura 0. Esquema general del sistema

Ugarit esta conformada por 3 (tres) grandes sistemas: el de la pantalla sensible al tacto, el analizador de grafías y generador de partituras, y por ultimo el compositor de música.

El sistema de pantalla sensible al tacto envía información sobre los dedos y sus eventos (presionar, arrastrar, levantar) al sistema analizador de grafías. Este interpreta sus movimientos y sus trayectos según los punteros (dedos) que recibe, a su vez discrimina que grafías son aceptadas como gestos musicales y cuales no. De este modo el generador de partituras va “escribiendo” una partitura que, una vez finalizada, es interpretada por el sistema encargado de la composición musical.

2. Soporte Físico

A continuación se detallará el proceso de diseño y construcción de la interface de pantalla sensible al tacto (del tipo multi-tacto) aplicada al editor gestual de música que es objeto del presente escrito.

2.1. Antecedentes

En etapas anteriores del presente proyecto PICTO se han desarrollado otros dispositivos de captación a modo de mesa táctil, buscando como objetivo obtener la mayor superficie de proyección y captación con la mayor economía de espacio. En los casos anteriores los dispositivos estaban orientados a aplicaciones que requerían un importante espacio de visualización de contenido gráfico, como es el caso de la aplicación “Mudo Circular”.

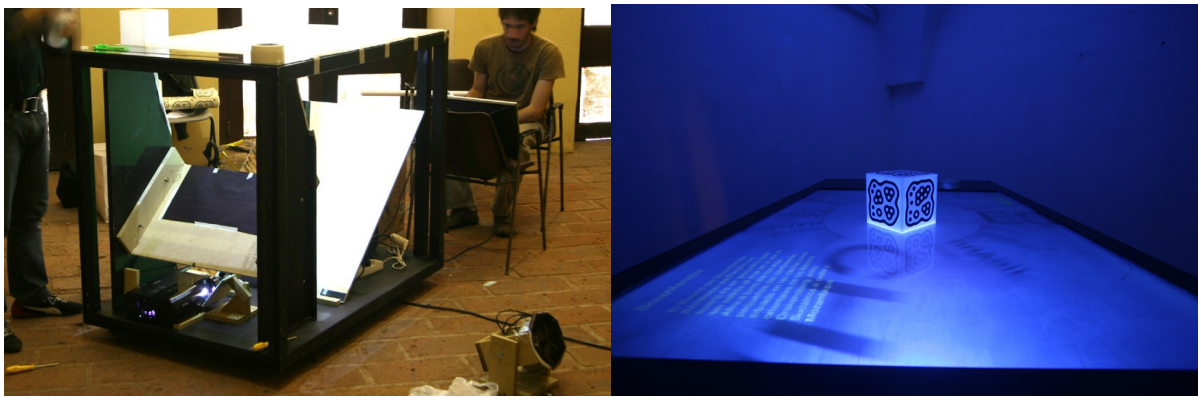

Figura 1. Instalación Mundo Circular

En cambio para el diseño de presente dispositivo, enmarcado en la etapa final del proyecto en donde se desarrolla un editor de partitura gestual, el diseño se orientó a crear una estructura de menor tamaño, que se centrara más en la captación que en la visualización, por lo que no requeriría una pantalla de gran tamaño.

Se buscó solucionar los siguientes problemas que presentó el prototipo anterior:

- El dispositivo no puede desarmarse completamente. Complejidad en el armado y traslado.
- Multiplicidad de componentes, el sistema de reflexión de 3 espejos resulta complejo en su instalación, requiere un espacio considerable y de una calibración y estabilidad extremas.
- El empleo de lámparas infrarrojas eleva la temperatura interna del dispositivos, obligando a utilizar ventilación.

2.2. Requisitos para el diseño del dispositivo

De acuerdo a lo anteriormente mencionado se definieron una serie de objetivos para la construcción de la estructura del dispositivo:

- El dispositivo debe tener un tamaño que permita ser fácilmente trasladado.
- En caso de ser necesario el desarme, presentar un diseño que economice el tiempo de montaje.
- Reducir la cantidad de componentes, empleando el mínimo de espejos.
- Buscar una alternativa a la utilización de lámparas infrarrojas para economizar espacio y evitar el calentamiento del dispositivo.

Una vez evaluados los objetivos, se consideró que el dispositivo, al cual llamaremos mesa táctil, podría no superar los 900 mm y que una pantalla de 550 x 420 mm cumpliría perfectamente con los requisitos del sistema de captación y permitiría una correcta interacción con la aplicación, como así también proporcionar la mínima altura necesaria para una gestualidad cómoda, en relación a la postura y el movimiento de las manos del usuario.

Al mismo tiempo se consideró continuar con la tecnología de captación óptica, con retro-proyección, solo que en esta oportunidad el dispositivo se ideó para contener solamente los dispositivos de entrada y salida, o sea el proyector, la cámara y el sistema de iluminación. Se imagino la estructura como una gran interface independiente, la cual se conectara de manera externa al ordenador. Parte de las dificultades y las grandes dimensiones de los prototipos anteriores se debían a que también contenían al ordenador. Se prefirió excluirlo de la estructura contenedora.

En último lugar se optó como alternativa a la iluminación de lámparas infrarrojas el empleo de leds infrarrojos, dispuestos directamente sobre la superficie de la pantalla.

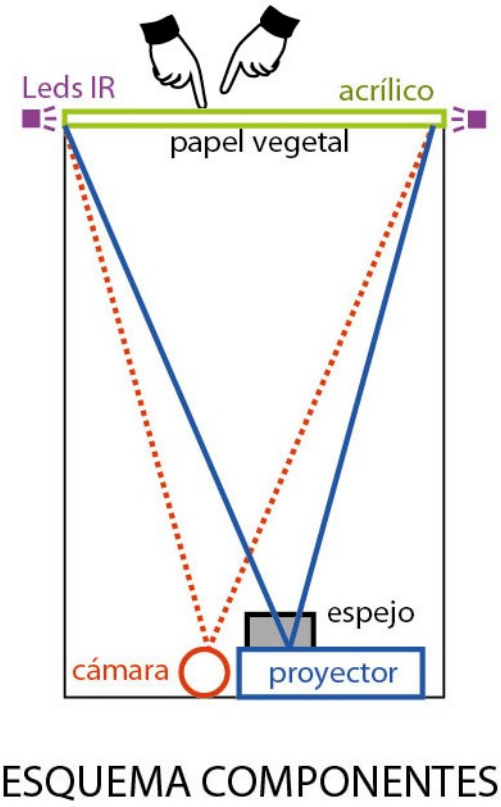

Figura 2. Sistema de captación

2.3. Construcción estructura contenedora

La estructura es simple, un paralelepípedo que tiene una dimensión total de 574 x 444 x 900 mm, dividido en 2 piezas principales, la pantalla bastidor que funciona a modo de tapa de encastre, de 100 mm de alto y el cuerpo del dispositivo de 840 mm de alto, resultando luego del ensamble en una altura total de 900 mm (40 mm de encastre).

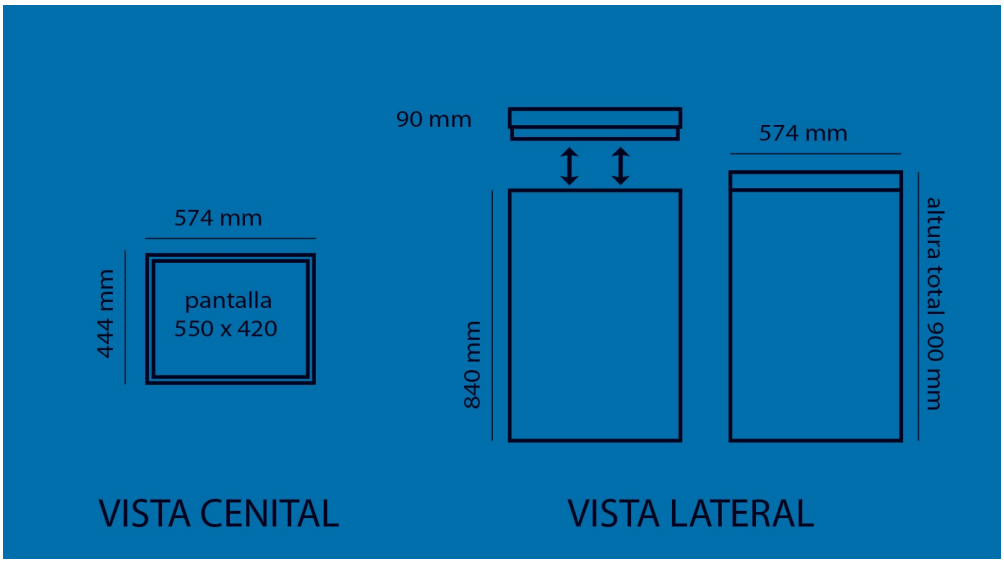

Figura 3. Vistas de la estructura

La mesa táctil construida con placas de madera MDF (Fibrofácil) espesor 9 mm, unidas y atornilladas entre si por soportes metálicos “L”.

Componentes Bastidor “tapa” de la pantalla:

- 2 placas de 556 x 60 mm
- 2 placas de 444 x 60 mm
- 2 placas de 426 x 90 mm
- 2 placas de 538 x 90 mm

Componentes Cuerpo de la estructura:

- 2 placas de 556 x 840 mm
- 2 placas de 444 x 840 mm
- 1 placa de 574 x 444 mm

La estructura resultante es fácilmente transportable, y si así fuera necesario el desarme es de igual manera sencillo, ya que solo requiriere por un lado la separación de la tapa superior (pantalla) y el resto de la mesa es desmontable desatornillando algunas de los soportes que unen las placas, para poder dividir la estructura.

El dispositivo posee una de sus caras laterales unida por medio de bisagras, lo que conforma una especie de puerta, que permite colocar y calibrar los componentes internos, cámara y proyector, como así también retirarlos durante el traslado, por un tema de seguridad, para evitar que se dañen dichos equipos.

Figura 4. Construcción de la mesa. Detalle del armado.

2.4. La superficie de la pantalla y los reflejos

Al momento de la selección de los materiales para la construcción de la pantalla (proyección y área táctil) se tuvo en cuenta la experiencia del prototipo anterior. Se emplearon los mismos materiales: acrílico y papel vegetal.

Recordemos que la pantalla de proyección, que a su vez funciona como superficie táctil, debe cumplir con los siguientes requisitos:

1. **Resistencia:** debe ser lo suficientemente resistente para permitir apoyar objetos y posar los dedos para interactuar (a esto se suma el hecho de que la altura de la mesa invita al público a apoyarse sobre esta).
2. **Opacidad**: debe ser lo suficientemente opaca para permitir ver nítidamente (desde abajo) sólo los objetos apoyados, pero no aquellos por encima. Por ejemplos, debe poder verse los dedos apoyados, pero no la mano por encima de ellos, ni las caras de las personas.
3. **Transparencia y nitidez:** también debe poder ser lo suficientemente transparente para dejar ver la imagen proyectada por detrás, pero lo suficientemente opaca como para atrapar la luz y no dejarla pasar. Por ejemplo: si es muy transparente, la proyección seguirá de largo (proyectándose en el techo, en vez de la pantalla) , si es muy opaca no podrá verse bien la imagen (hasta puede perder nitidez).
4. **Antireflejo:** la superficie debajo de la pantalla debe poseer un tratamiento anti-reflejo ya que el sistema de captura es muy sensible a los altos contrastes, principalmente si los producen los reflejos en este superficie. Cuando la superficie no es anti-reflejos, funciona como un espejo que refleja el sistema de iluminación.

Por lo tanto se consideró que la combinación de una placa de acrílico y papel vegetal podían cumplir los requisitos de opacidad y transparencia al mismo tiempo. A diferencia del prototipo anterior, el

papel vegetal en este caso se colocó por debajo del acrílico, adherido perimetralmente con cinta bifaz, lo que permitió cumplir con los requisitos de resistencia en la cara externa y de superficie anti-reflejo en la cara interna. Debemos especificar que el tipo de acrílico utilizado es uno especial, comercializado como ACRYLITE® EndLighten, el cual posee propiedades que mejoran la refracción interna de la luz y su elección se correspondió con la utilización de los leds infrarrojos que se detallaran en el apartado de iluminación.

2.5. Ubicación de la cámara y proyector

No hubo inconvenientes en relación a la ubicación de la cámara y el proyector. Por las dimensiones de la pantalla y la estructura, la distancia de 900mm desde la base del dispositivo hasta la pantalla resulta adecuada tanto para la realización de la proyección como de la captación de la cámara. El proyector se dispone sobre la base de la mesa, y se utiliza solo un pequeño espejo a 45º para dirigir el haz de luz hacia la pantalla de manera perpendicular a ésta. Es por esto mismo, por la simplicidad del sistema de proyección, que se dispone de suficiente espacio para colocar la cámara en el piso del dispositivo al lado del proyector, sujeta a la base y dispuesta directamente en 90º apuntando hacia arriba.

Figura 5. Disposición de los dispositivos de proyección y captación.

2.6. Iluminación

La iluminación que este sistema utiliza es infrarroja. En esta instancia, se decidió utilizar Leds infrarrojos, más duraderos y como alternativa a lámparas incandescentes filtradas con acetatos que ocupan espacio y generan un calentamiento de la estructura.

Los LEDs son de tipo SMD 3528 (de 3.5mm x 2.8mm) y vienen pre-ensamblados sobre un material flexible gracias al cual es posible la adaptación al bastidor de la pantalla. Una de sus caras cuenta con adhesivo 3M, permitiendo colocar las tiras de forma fácil y rápida sobre el marco, iluminando directamente sobre el canto del acrílico.

Como se ha mencionado anteriormente el acrílico empleado es el ACRYLITE® EndLighten, un acrílico semitransparente con propiedades especiales para conducir y difundir la luz, razón por la cual la difusión y refracción interna de los rayos infrarrojos se ven potenciadas. La luz de los leds IR ingresa sobre el borde de la placa generando una distribución uniforme de la luz sobre la superficie.

Figura 6. Iluminación infrarroja.

3. Generación de partituras según grafías musicales

La aplicación para la generación de partituras según grafías musicales se desarrollo en Processing, un lenguaje basado en Java desarrollado por Fly y Reas (vinculados al M.I.T. y la U.C.L.A), el cual fue diseñado para contemplar las necesidades de los artistas relacionados con las nuevas tecnologías.

La aplicación cuenta con 4 (cuatro) grandes partes:

- Un mediador entre reactivision y la aplicación en si, esto nos permite corregir errores ópticos, generados como la cámara web, como puede ser el ruido.
- Una interface la cual nos permite interactuar con la pantalla sensible al tacto.
- Un analizador e interprete de grafías, quien nos permite analizar los trayectos de los punteros e interpretar a que grafía musical corresponde.
- Un generador de partituras, este es el encargado de, una vez finalizada la composición, generar un archivo con un formato determinado, el cual sera interpretado por Pure Data.

Cada una de estas partes esta relacionada y en cada generación de partituras vuelven a iniciarse, en resumidas cuentas la aplicación resuelve niveles de ruidos generados por el sistema de visión artificial, se encarga de traducir datos cartesianos en parámetros perceptivos, esto es poder discriminar si una linea es zigzageante, horizontal o vertical, por ultimo genera una partitura normalizando sus valores a escalas musicales.

3.1. Diagrama de objetos o clases

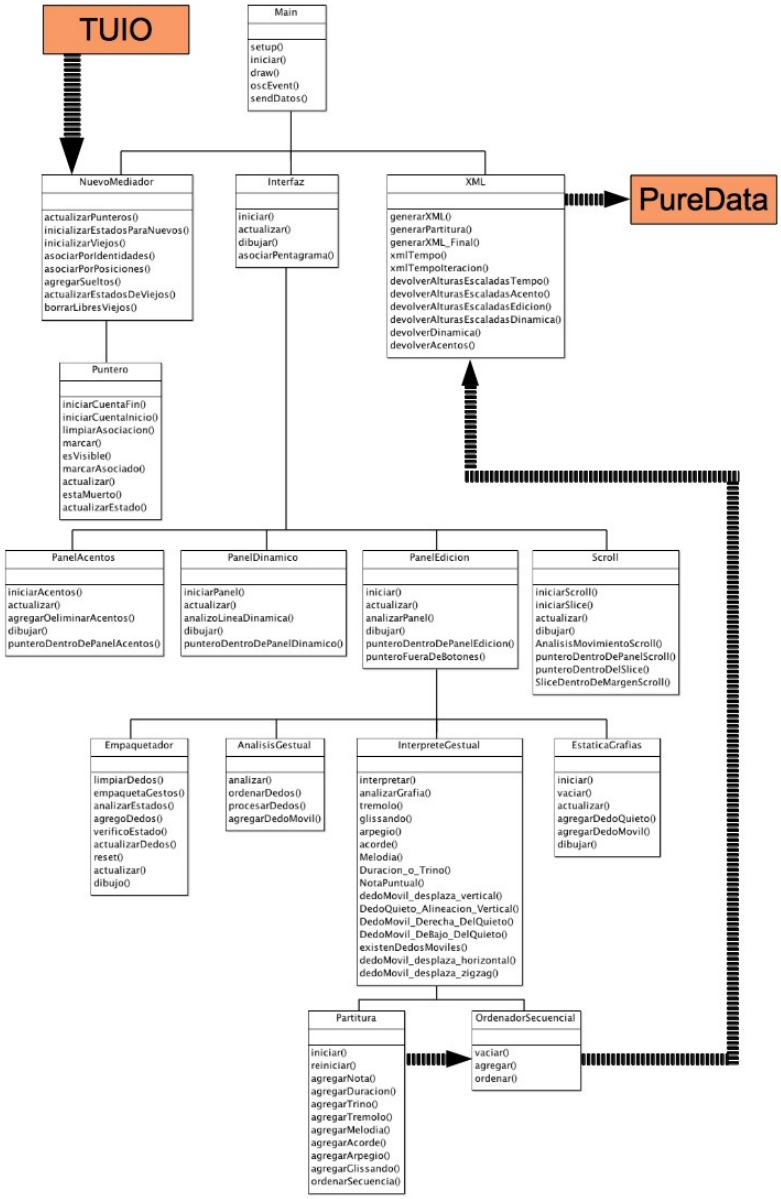

Figura 7. Diagrama de objetos.

El imagen anterior muestra un diagrama detallado de los objetos que entran en juego en la aplicación, y como se relacionan entre si, algunos de ellos son:

- **NuevoMediador**: sirve de intermediario entre el objeto que interpreta el protocolo TUIO y el resto de la aplicación. Particularmente sirve para corregir los errores y el ruido que el cliente (de visión artificial) destinado a la captación de los dedos pueda generar.

- **Interfaz**: se encarga de organizar la interfaz gráfica de la aplicación, dándole a cada región de la pantalla su correspondiente funcionalidad. La misma se encuentra conformada por diferentes paneles: edición, tempo, dinámica, acento y scroll.
- **Empaquetador**: es el encargado de ir almacenando temporalmente los dedos ingresados en la pantalla. Registra su evolución en el tiempo cual si fueran punteros. Clasifica los dedos en estáticos o móviles.
- **Análisis Gestual**: es el encargado de analizar los diferentes tipos de movimientos que puede generar un dedo en el desarrollo de un gesto, este análisis incluye generación de descripciones respecto a la rectitud o zigzageos de los trazos, la dirección, la curvatura, así como su posición relativa en función de los demás dedos.
- **Interprete Gestual**: recibe la descripción hecha por el análisis gestual pero a a diferencia de este hace un análisis integral del comportamiento en conjunto de todos los dedos en la configuración de un "gesto", se encarga de analizar cuando empieza y termina un "gesto", así como de revisar si estos cumplen con "una buena forma".
- **Estética Grafías**: se encarga de redibujar reemplazando los gestos aceptados por sus correspondientes grafías musicales. En definitiva, son la representación de la transformación del gesto en símbolo.
- **Partitura**: se encarga de recibir y almacenar los elementos musicales interpretados a parir de los gestos. Luego el objeto "Ordenador Secuencial" se encarga de organizar dichos elementos en la linea de tiempo para la "interpretación musical" por parte de Pure Data.

A continuación se explicaran cada uno de los objetos detalladamente, ilustrando sus relaciones y funcionamientos internos.

3.2. Mediador

El objeto **"NuevoMediador"** sirve como intermediario entre el objeto que interpreta el protocolo TUIO y el resto del programa. Particularmente sirve para corregir los errores y el ruido que el cliente (de visión artificial) destinado a la captación de dedos pueda generar.
El funcionamiento interno del objeto esta dado por dos grandes partes, por un lado un objeto TuioProcessing (heredado del protocolo TUIO) quien nos suministra información sobre la cantidad de dedos que hay en la pantalla, sus coordenadas cartesianas (x e y), sus velocidades de desplazamiento, sus rotaciones, entre otros. Por otro lado, una instancia de un ArrayList, donde se van a ir almacenando de manera temporal objetos de tipo Puntero (dedos), los cuales van a ser procesados en cada iteración del programa por una serie de algoritmos capaces de solucionar los errores y el ruido que el cliente pueda generar.

La clase tiene un solo constructor:

```
NuevoMediador( TuioProcessing mensajero_ ) {
   mensajero = mensajero_;
   listaViejos = new ArrayList();
}
```

como puede verse el constructor **"NuevoMediador"** recibe una instancia del objeto TuioProcessing llamada mensajero y crea un ArrayList llamado "listaViejos", esto se ejecuta una sola vez al iniciar el programa.

Los métodos o subrutina (subalgoritmo que forma parte del algoritmo principal, el cual permite resolver una tarea específica) que dan funcionamiento a dicho objeto son los siguientes:

- inicializarEstadosParaNuevos();
- asociarPorIdentidades();
- asociarPorPosiciones();
- AgregarSueltos();
- ActualizarEstadosDeViejos();
- borrarLibresViejos();

con el nombre de los métodos ya tenemos una aproximación de la tarea que tiene asignado cada uno, igualmente cabe aclarar que éstos son solo los principales ya que el objeto cuenta con más métodos, a continuación se detallaran y explicar los algoritmos de cada uno de ellos.

```
void inicializarEstadosParaNuevos() {

    listaCursor = mensajero.getTuioCursors();

    cantPuntNuevos = listaCursor.size();

    estadoNuevos = new String[ cantPuntNuevos ];

    for( int i=0 ; i<cantPuntNuevos ; i++ ) {

      estadoNuevos[i] = "pendiente";

    }
}
```

Como puede verse la variable cantPuntNuevos almacena la cantidad de dedos que hay sobre la mesa, luego esta cantidad es utilizada para crear un array de tipo String y por ultimo se recorren todos los dedos a través de un ciclo **for** y se le asigna el estado de **"pendiente"**.
El asociarle el estado **"pendiente"** a un dedo que recién ingresa en la pantalla es útil ya que el sistema óptico de captura, en este caso una cámara web, suele generar ruido entre fotograma y fotograma lo que lleva a que aparezcan lo que se denomina "dedos fantasmas", por esta misma razón se le asocia a todos los dedos dicho estado para poder analizarlos y corroborar que pasado determinado umbral de tiempo sigan existiendo, si esto es así se lo considera un "dedo real" y sino se lo descarta puesto que es un **"dedo fantasma"**.

```
void asociarPorIdentidades() {

    for( int j=0 ; j<listaViejos.size() ; j++ ) {

    Puntero este = (Puntero) listaViejos.get( j );

      if( !este.asociado ) {

        for (int i=0 ; i<cantPuntNuevos ; i++) {

          if( estadoNuevos[i].equals( "pendiente" ) ) {

            TuioCursor unCursor = (TuioCursor) listaCursor.elementAt(i);

            if( unCursor.getSessionID() == este.idTuio ) {

              este.actualizar( unCursor );
```

```
              este.marcarAsociado();
              estadoNuevos[i] = "ocupado";
              break;

            }
          }
        }
      }
    }
  }
```

El algoritmo anterior se encarga, como su nombre lo indica, de asociar a los dedos por su identidad (ID), esto significa recorrer todos los dedos que han estado presentes en el fotograma 1, verificar que no tengan ya un dedo asociado, si esto es así se recorren todos los dedos del fotograma 2 consultando si su estado es **"pendiente",** si encuentra alguno con tal estado compara que su id de sesión sea igual al del fotograma anterior:

```
if( unCursor.getSessionID() == este.idTuio ) {

  este.actualizar( unCursor );
  este.marcarAsociado();
  estadoNuevos[i] = "ocupado";
  break;

}
```

Pasada dicha verificación actualiza la posición del dedo, lo marca como asociado y modifica su estado de **"pendiente"** a **"ocupado"**. En otras palabras, esta función es la primera validación por la que pasan los dedos para descargar la existencia de **"dedos fantasmas"**.

```
void asociarPorPosiciones() {

    for( int j=0 ; j<listaViejos.size() ; j++ ) {

      Puntero este = (Puntero) listaViejos.get( j );

      if( !este.asociado ) {

        for (int i=0 ; i<cantPuntNuevos ; i++) {

          if( estadoNuevos[i].equals( "pendiente" ) ) {

            TuioCursor unCursor = (TuioCursor) listaCursor.elementAt(i);

            if( estanCerca( este, unCursor ) ) {

              este.actualizar( unCursor );
              este.marcarAsociado();
              estadoNuevos[i] = "ocupado";
```

```
                break;

              }
            }
          }
        }
      }
    }
```

El método “asociarPorPosiciones”, al igual que el método “asociarPorIdentidades”, se encarga de descartar la existencia de **“dedos fantasmas”** , en este caso comparando las coordenadas cartesianas en la que se encuentra un dedo en el fotograma 1 y 2, esto se comprueba a través de la siguiente función:

```
boolean estanCerca( Puntero este, TuioCursor unCursor ) {
    return dist( este.x, este.y,
    unCursor.getScreenX(width), unCursor.getScreenY(height) ) <
toleranciaDistancia;
  }
```

Acá puede verse como recibe como parámetros un objeto Puntero (dedo almacenado en fotograma 1) y un objeto TuioCursor (dedo existente en el fotograma 2) y se los analiza a través de la función dist(x1 , y1 , x2 , y2) la que nos retorna la distancia existente entre los dos puntos, la cual se compara si es mayor o menor a la variable global **“toleranciaDistancia”.** Si el valor que nos devuelve es TRUE actualizamos los dedos, lo marcamos como asociado y cambiamos su estado a **“ocupado”**.

Una aclaración importante es el porque la existencia de la variable **“toleranciaDistancia”**, la misma se utiliza para no ingresar punteros generados por el movimiento inconsciente del dedo sobre la pantalla, esto quiere decir que si el usuario apoya su dedo en la pantalla y el mismo se mueve 1px el sistema lo reconocería como un nuevo puntero, por esta razón la existencia de dicha variable.

La siguiente imagen ilustra lo explicado anteriormente, es decir, como se generan nuevos punteros a pesar de que el movimiento de los dedos fue muy sutil.

Figura 8. Niveles de Tolerancia.

En otras palabras, **"toleranciaDistancia"** es el valor en pixeles que tiene que moverse un dedo entre fotograma 1 y 2 para ser considerado movimiento.

Una vez analizados los dedos con los métodos "asociarPorPosiciones" y "asociarPorIdentidades" se procede a agregar a aquellos que quedaron pendientes de los nuevos.

```
void agregarSueltos() {

    for( int i=0 ; i<cantPuntNuevos ; i++ ) {

      if( estadoNuevos[ i ].equals( "pendiente" ) ) {

        TuioCursor unCursor = (TuioCursor) listaCursor.elementAt(i);
        Puntero este = new Puntero( unCursor );
        listaViejos.add( este );
        estadoNuevos[ i ] = "nuevo";

      }
    }
  }
```

Esto quiere decir recorrer todos los dedos recibidos por el protocolo TUIO, verificar uno por uno si su estado es **"pendiente"**, si esto es así se crea un objeto de tipo Puntero y se lo agrega a "listaViejos" y cambia su estado a **"nuevo"**.

Hasta el momento hemos ido creando objetos Punteros y asignándoles estados según su condición sobre la pantalla, llegado este punto es necesario actualizar dichos estados a través de una *MEF* (maquina de estados finitos) que se encuentra en el objeto Puntero (explicado en la sección Objeto Puntero).

```
void actualizarEstadosDeViejos() {

    for( int i=0 ; i<listaViejos.size() ; i++ ) {

      Puntero este = (Puntero) listaViejos.get( i );
      este.actualizarEstado();

    }
  }
```

Por ultimo nos queda borrar de “listaViejos” los dedos que ya no se encuentran en la pantalla, para tal proceso se utiliza la siguiente función:

```
void borrarLibresViejos() {

   cantPunteros = 0;

    for( int i=0 ; i<listaViejos.size() ; i++ ) {

      Puntero este = (Puntero) listaViejos.get( i );

      if(este.esVisible()) {

        cantPunteros++;

      }

      else if( este.estaMuerto() ) {

        listaViejos.remove(i);

      }

    }

}
```

Para este proceso es necesario tener en cuenta un margen de tolerancia para borrar el dedo de la pantalla, esto quiere decir que si en el fotograma 1 ingresa un dedos, en el fotograma 2 se levanta o el sistema óptico no lo detecta y en el fotograma 3 vuelve a ingresar o se vuelve a detectar se lo siga considerando como el mismo dedo y no como un dedo nuevo. Por esta misma razón se utiliza la siguiente condición:

```
if(este.esVisible()) {

  cantPunteros++;

}
 else if( este.estaMuerto() ) {

   listaViejos.remove(i);

 }
```

En otras palabras, en valor que se le asigne a la **"tolerancia"** en la cantidad de fotogramas en los que no se tiene que encontrar presente el dedo para ser borrado de la lista **"listaViejos"**.

3.2.1. Puntero

El objeto **"Puntero"** es quien representa de manera matemática y lógica a los dedos "reales" que están sobre la pantalla, esto quiere decir que por cada dedo que el objeto **"NuevoMediador"** reconozca y verifique su existencia se creara un objeto **"Puntero",** quien cuenta con una *MEF* (maquina de estado finito) para verificar su existencia (estados) a lo largo del tiempo. En otras palabras el objeto tendría –por decirlo de algún modo- el funcionamiento de un cursor en pantalla.

El objeto cuenta con un solo constructor:

```
Puntero( TuioCursor unCursor ) {

    idTuio = unCursor.getSessionID();

    id = contId;

    contId = (contId+1) % 1000;

    x = unCursor.getScreenX(width);

    y = unCursor.getScreenY(height);

    down = false;

    up = false;

    delete = false;

    estado = "dudoso";

    iniciarCuentaInicio();

    cuentaInicio = toleranciaInicio;
```

```
    asociado = true;

}
```

Como puede verse en el código anterior para crear una instancia de **"Puntero"** se necesita pasarle como parámetro un objeto TuioCursor, heredado del protocolo TUIO, quien nos suministra información sobre el estado de los dedos en pantalla. De este modo seteamos las variables iniciales del objeto (solo se explican las principales):

1. **idTuio:** es la variable que define en que sesión (periodo de una conexión en la cual se intercambian paquetes de datos) se encuentra el dedo en pantalla.

2. **id:** identificador numérico del puntero en el sistema.

3. **x:** posición del dedo en en la pantalla en el eje de las abscisas.

4. **y:** posición del dedo en en la pantalla en el eje de las ordenadas.

5. **down, up y delete:** sirven para saber el estado del dedo, si esta presionado, levantado o ha sido borrado, en este caso todo esta seteado en FALSE.

6. **estado:** variable que almacena el estado del dedo al ingresar en pantalla, como aun no se sabe si es un "dedo fantasma" se lo setea como "dudoso".

Cabe aclarar que este objeto es comandado por el objeto **"NuevoMediador"**, esto quiere decir que sus métodos son ejecutados por este ultimo.

Como se explico en el apartado de "Objeto NuevoMediador" para que un dedo sea reconocido y aceptado tenia que pasar cierto umbral de tiempo (fotogramas), lo mismo para que dicho dedo dejara de existir, esas variables de umbral son seteadas dentro de este objeto por medio de los siguientes métodos:

```
int toleranciaInicio = 10;

int toleranciaFin = 10;

...

void iniciarCuentaFin() {

  cuentaFin = toleranciaFin;

}

void iniciarCuentaInicio() {

  cuentaInicio = toleranciaInicio;

}

...
```

El poder setear las variables **toleranciaInicio** y **toleranciaFin** nos permite tener mas control sobre el estado de los dedos en pantalla, esto es, si nuestro sistema de captura óptico emite mucho ruido lo mas probable es que tengamos que aumentar los valores de tolerancia para poder corregirlos.

Una vez reconocidos los dedos es preciso poder actualizar sus posiciones, este proceso es llevado a cabo por la siguiente función:

```
void actualizar(TuioCursor esteCursor) {

    idTuio = esteCursor.getSessionID();

    x = esteCursor.getScreenX(width);

    y = esteCursor.getScreenY(height);

}
```

La función recibe como parametro a una instancia del objeto TuioCursor y de este modo podemos actualizar las variables **idTuio** , **x** e **y** respectivamente.

Una vez actualizados es preciso poner en funcionamiento la *MEF* (maquina de estados finitos) quien nos permite analizar los diferentes periodos de “vida” de cada uno de los dedos.

```
void actualizarEstado() {

    if( estado.equals("dudoso") ) {
      cuentaInicio--;
      if( !asociado ) {
        estado = "muerto";
      }
      else if( cuentaInicio<=0 ) {
        estado = "activo";
        down = true;
        up = false;
      }
    }
    else if( estado.equals("activo") ) {
      down = false;
      up = false;
      if( !asociado ) {
        estado = "moribundo";
        iniciarCuentaFin();
      }
    }
    else if( estado.equals("moribundo") ) {
      cuentaFin--;
      if( asociado ) {
        estado = "activo";
      }
      else if( cuentaFin<=0 ) {
        estado = "sin_retorno";
        up = true;
      }
    }
    else if( estado.equals("sin_retorno") ) {
      estado = "muerto";
    }
    else if( estado.equals("muerto") ) {
    }
    else {
      println("error!!!!!);
    }
  }
```

El algoritmo anterior lo que esta haciendo es poner al dedo a prueba constantemente a través de cuatro grandes pruebas:

Prueba 1:

```
if( estado.equals("dudoso") ) {
      cuentaInicio--;
      if( !asociado ) {
        estado = "muerto";
      }
      else if( cuentaInicio<=0 ) {
        estado = "activo";
        down = true;
        up = false;
      }
}
```

Cuando la variable **estado** se encuentra en "**dudoso**" la variable cuentaInicio, seteada inicialmente con el valor de 10, empieza a restar sus valores mientras pregunta si no tiene un dedo asociado, si esto se cumple considera que el dedo dejo de existir y pone su estado en "**muerto**". Ahora bien, si tiene un dedo asociado y cuentaInicia llega a tener un valor menor o igual a 0 considera que el dedo esta "**activo**" esto es que ha sido presionado y aun no se ha levantado, por eso mismo se setea las variables **down** en TRUE y **up** en FALSE.

Prueba 2:

```
else if( estado.equals("activo") ) {
      down = false;
      up = false;
      if( !asociado ) {
        estado = "moribundo";
        iniciarCuentaFin();
      }
}
```

Cuando la variable **estado** toma el valor "**activo**" se setean a las varia **down** y **up** en FALSE, esto quiere decir que el dedo ya ha sido presionado pero aun no se ha levantado, nuevanmente se pregunta sino tiene un dedo asociado, si esto es así la variable estado pasa a tener el valor

“**muribundo**” y activa la funcion **iniciarCuentaFin();** quien internamente setea la variable cuentaFin en 10, esto se hace para tener un umbral de tiempo en el que el dedo puede llegar a aparecer.

Prueba 3:

```
else if( estado.equals("moribundo") ) {
      cuentaFin--;
      if( asociado ) {
        estado = "activo";
      }
      else if( cuentaFin<=0 ) {
        estado = "sin_retorno";
        up = true;
      }
}
```

Cuando **estado** toma el valor “**moribundo**” la variable cuentaFin empieza a disminuir su valor si esta llega a 0 estado pasa a tomar el valor “**sin_retorno**” y la variable **up** se setea en TRUE, esto quiere decir que el dedo ha sido levantado. Mientras cuentaFin no este en 0 se pregunta si el dedo tiene un asociado si esto se cumple estado vuelve a tomar el valor “**activo**”.

Prueba 4:

```
else if( estado.equals("sin_retorno") ) {
  estado = "muerto";
}
```

Como ultimo si el **estado** llego a estar en “**sin_retorno**” se considera al dedo como “**muerto**”, esto es que dejo de existir definitivamente en la pantalla.

Como puede verse el manejar diferentes **estados** para un dedo nos permite ser mas cautelosos a la hora de discriminar la existencia de uno, esto sirve para evitar la presencia de “dedos fantasma” y a su vez nos permite un mejor seguimiento del dedo a lo largo del tiempo.

3.3. Interfaz

El objeto “Interfaz” se encarga de organizar la gráfica de la aplicación dándole a cada región de la pantalla su correspondiente funcionalidad. La misma esta confrontada por los diferentes paneles:

- Edición: región de la pantalla donde se dibujan las grafías musicales.
- Tempo: región de la pantalla donde se manipula el tempo de la melodía.
- Dinámica: región de la pantalla donde se manipula la dinámica de la melodía.
- Acento: región de la pantalla donde se marcan los acentos de la melodía.
- Scroll: región que nos permite deslizarnos a lo ancho de la pantalla, posibilitándonos una mayor dimensión para el dibujo de la melodía.

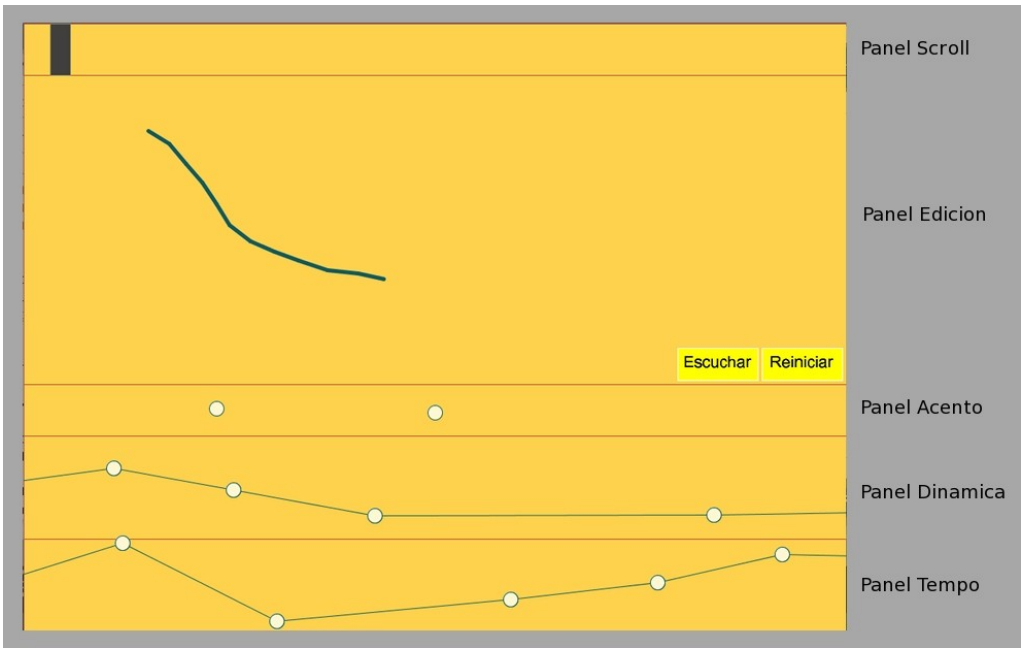

Figura 9. Interface de la aplicación.

Internamente este objeto no tiene gran complejidad puesto que solo se encarga de inicializar los paneles (objetos) en sus respectivas posiciones, darle el color correspondiente a los mismos y de su respectiva actualización.

Para poder setear el color de la interfaz solo basta con modificar las siguientes variables:

```
  color borde = color(#bc4831);
  color rellenoActivo = color(#da9d35);
  color rellenoPasivo = color(#fed24d);
```

Como su nombre indica **borde** se encarga de almacenar el color de los bordes de los paneles, **rellenoActivo** contiene el color de los paneles cuando están siendo presionados y por ultimo **relenoPasivo**, contrario al anterior, cuando no hay ningún dedo sobre ellos.

El metodo encargado de inicializar los paneles es el siguiente:

```
void iniciar() {

    panelEdicion = new PanelEdicion(escenario);
    panelEdicion.dimenciones(0,altoStage/12,anchoStage,altoStage/2);
    panelEdicion.escalaEjeY(127,24);
    panelEdicion.paleta(borde,rellenoActivo,rellenoPasivo);

   //-----------------------------------

    panelAcentos = new PanelAcentos(escenario);
panelAcentos.dimenciones(0,altoStage/12+altoStage/2,anchoStage,altoStage/12);
    panelAcentos.escalaEjeY(20,20);
    panelAcentos.paleta(borde,rellenoActivo,rellenoPasivo);
```

```
    //---------------------------------------

    panelDinamica = new PanelDinamico(escenario);
panelDinamica.dimenciones(0,altoStage/2+altoStage/12+altoStage/12,
anchoStage,altoStage/6);
    panelDinamica.escalaEjeY(0,107);
    panelDinamica.paleta(borde,rellenoActivo,rellenoPasivo);

   //---------------------------------------

    panelTempo = new PanelDinamico(escenario);
panelTempo.dimenciones(0,altoStage/2+altoStage/12+altoStage/12+altoStage/6,
anchoStage,altoStage/6);
    panelTempo.escalaEjeY(maxValorTempo,minValorTempo);
    panelTempo.paleta(borde,rellenoActivo,rellenoPasivo);

    //---------------------------------------

    scrollHorizontal = new
Scroll(escenario);scrollHorizontal.dimenciones("HORIZONTAL",0,0,
anchoStage,altoStage/12);
    scrollHorizontal.paleta(borde,rellenoActivo,rellenoPasivo);

    //---------------------------------------

    borrar = new Boton("Reiniciar",(anchoStage-100)-margenBotones,
((altoStage/12+altoStage/2)-40)-margenBotones);
 borrar.paleta(color(255,255,255),color(255,0,255),color(255,255,0));

   //---------------------------------------

    escuchar = new Boton("Escuchar",(anchoStage-200)-(margenBotones*2),
((altoStage/12+altoStage/2)-40)-margenBotones);
escuchar.paleta(color(255,255,255),color(255,0,255),color(255,255,0));

  }
```

Como puede verse el codigo se encarga de inicilizar cada uno de los objetos, esto incluye:

1. Setear las dimenciones y posicion de donde queremos colocar el panel.
2. Setear la paleta de color por medio de las variables explicadas anteriormente.
3. Setear su escala en eje Y, esto se utiliza para convertir los pixeles a escalas musicales.

El método que se encarga de dibujar fotograma a fotograma los cambios que en cada panel es el siguiente:

```
void actualizar(NuevoMediador mediadorInterprete_) {

    mediadorInterprete = mediadorInterprete_;

    panelEdicion.actualizar(mediadorInterprete);

    panelDinamica.actualizar(mediadorInterprete);

    panelTempo.actualizar(mediadorInterprete);
```

```
    panelAcentos.actualizar(mediadorInterprete);

    scrollHorizontal.actualizar(mediadorInterprete);

    borrar.actualizar(mediadorInterprete);
    if (borrar.botonActivo()) {
      reIniciar = true;
      borrar.resetear();
    }

    escuchar.actualizar(mediadorInterprete);
    if (escuchar.botonActivo()) {
      xml = new XML(this);
      xml.generarPartitura("xml/partitura.xml");
      sendDatos(1); //envia por OSC reproducir
      escuchar.resetear();
    }
  }
```

El método **actualizar** se ejecuta en cada iteración del programa recibiendo los datos del objeto NuevoMediador y actualizando internamente a cada panel.

Por otro lado, se encarga de ejecutar las acciones correspondientes a los botones “borrar” y “escuchar”, el primero se encarga de limpiar la pantalla en caso de confundirnos a la hora de dibujar las grafias o bien para volver a iniciar la aplicación. El segundo, se encarga de crear la partitura (archivo XML) y enviar un mensaje OSC (open sound control) al software PureData para que pueda empezar a interpretar la partitura.

3.3.1. Panel Acentos

El objeto “**PanelAcentos**” es quien nos permite agregarle acentos a la partitura que creamos, esto implica poder dibujar pequeños círculos -dentro de una región determinada- al apoyar el dedo, si se vuelve a apoyar el dedo sobre un acento ya echo este se elimina, de este modo podemos agregar y quitar acentos del panel sin interferir en el resto de la aplicación.

Este objeto se encarga de tres funciones básicas:

1. Crear región en pantalla para el panel.
2. Analizar si un dedo se encuentra dentro del panel.
3. Agregar o eliminar un acento.

El método **iniciar** nos permite configurar la posición y tamaño del panel en pantalla:

```
int x,y;

int ancho,alto;


…

void iniciar(int x_,int y_,int ancho_, int alto_) {
```

```
    x = x_;

    y = y_;

    ancho = ancho_;

    alto = alto_;

}

…

void dibujar() {
    pushMatrix();
    fill(relleno);
    stroke(borde);
    translate(x,y);
    rect(0,0,ancho,alto);
    popMatrix();
    dibujoAcento();
}

…
```

Como pude verse los atributos del método **iniciar** son asignados a las variables globales **x, y, ancho, alto** declaradas previamente, de esta manera podemos acceder a ellas desde cualquier método del objeto como puede ser el caso de la función **dibujar**.

En cada iteración del programa se debe actualizar el objeto **PanelAcento**, para así poder recibir que dedos hay en la pantalla y poder realizar sus cálculos, análisis y de este modo poder verificar si existen dedos en la región, si se están agregando o eliminando acentos.

```
…
void actualizar(NuevoMediador pantalla) {
    if(pantalla.cantPunteros > 0) {
      for(int i=0;i<pantalla.cantPunteros;i++) {
        Puntero este = pantalla.devolverPunteroNum(i);
        if(este != null) {
          if( punteroDentroDePanelAcentos(este)) {
            activoRegion = true;
            agregarOeliminarAcentos(este);
          }
          else {
            activoRegion = false;
          }
        }
      }
    }
    else {
      activoRegion = false;
    }
    cambioEstado(activoRegion);
```

```
  }
…
```

La función anterior se encarga de recibir un objeto **NuevoMediador** y verificar primeramente si existen dedos en la pantalla con **pantalla.cantPunteros > 0** , paso seguido recorre todos los dedos y consulta uno por uno si se encuentra dentro de la region, esto por medio del método **punteroDentroDePanelAcento**, si esta condicion se cumple se cambia el valor de la variable activoRegion y activa la función **agregarOeliminarAcentos**.

La funcion **punteroDentroDePanelAcento** nos retorna un **boolean**, **true** si el dedo esta dentro de la región o **false** si no:

```
…

boolean punteroDentroDePanelAcentos(Puntero estePuntero) {
    float x_ = estePuntero.x;
    float y_ = estePuntero.y;

     boolean valor = (x_ > x) && (x_ < (x+ancho)) && (y_ > y) && (y_ <
(y+alto));

    return valor;
  }

…
```

Para obtener esta información el algoritmo recibe como parámetro un objeto Puntero y se consulta si **estePuntero.x , estePuntero.y** se encuentra dentro de la región del panel por medio del siguiente calculo:

```
boolean valor = (x_ > x) && (x_ < (x+ancho)) && (y_ > y) && (y_ <
(y+alto));
```

En la siguiente ilustración puede verse lo antes explicado, o sea como la variable **EstadoRegion** cambia de valor según si el dedo ingresa o no a la región:

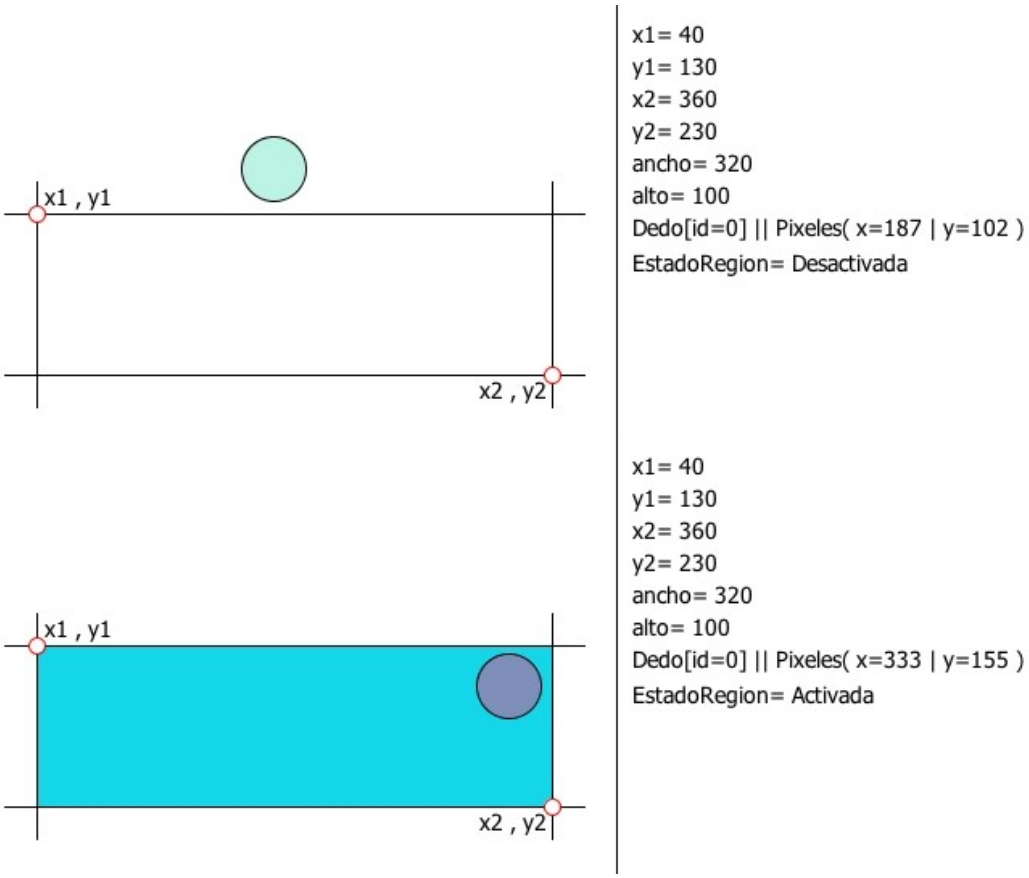

Figura 10. Regiones activas.

Dado el caso de que la región es activada es preciso verificar si hay que ingresar o eliminar un acento, tal proceso es llevado a cabo por el siguiente algoritmo:

```
int distanciaColision = 30;


…

void agregarOeliminarAcentos(Puntero estePuntero) {
    float x_ = estePuntero.x;
    float y_ = estePuntero.y;
    if(estePuntero.down) {
      boolean tocoAlguno = false;
      for(int j=acentos.length-1; j> 0; j--) {
        if(dist(escenario.tics_to_x(acentos[j].x), acentos[j].y, x_, y_) <=
distanciaColision) {
          acentos = (Acento[]) extraeDeArray(acentos, j);
          tocoAlguno = true;
          break;
        }
      }
      if(!tocoAlguno) {
        acentos = (Acento[])append(acentos,new
Acento(escenario.x_to_tics(x_),y_));
      }
    }
  }
```

```
...
```

Como puede verse se declara una variable **distanciaColision** con el valor de 30, ésta es la distancia en pixeles que tiene que haber entre un acento y otro. Para saber si hay una colisión entre acentos se recorre un array que contiene los acentos ya ingresados y compara sus posiciones con la del acento que se desea ingresar, si la distancia es menor se interpreta que el usuario intenta borrar el acento y se lo elimina del array que los contiene, sino toca ningún acento ya existente agrega uno nuevo.

3.3.2. Panel Dinamico

El objeto "**PanelDinamico**" es utilizado por la aplicación para crear dos paneles, por un lado el panel de la dinámica y por el otro del tempo, esto es así puesto que ambos paneles cumplen con la misma función a nivel aplicación/software pero diferente a nivel partitura, por lo que se opto por tener un solo objeto y crear dos instancias del mismo.

El objeto se caracteriza por tener una "linea dinámica" de manera horizontal en el centro de la región:

Figura 11. Linea dinámica sin quebrar.

ésta puede ser "quebrada" al ser presionada y de este modo se va graficando la dinámica o tempo de la partitura:

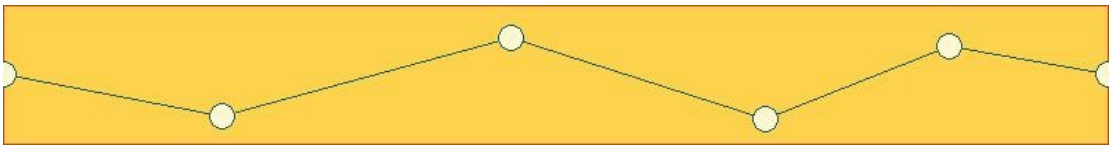

Figura 12. Linea dinámica quebrada.

Una característica importante de este objeto es el auto-ordenamiento de los nodos ingresados, para comprender este concepto hay que entender como funciona una matriz y como se lo recorre para poder armar la "linea dinámica" con sus respectivos nodos.

En principio, se puede considerar a una matriz o vector como una zona de almacenamiento continuo, que contiene una serie de elementos del mismo tipo, en nuestro caso son de tipo "PuntoDinamico" que es un objeto creado por nosotros.

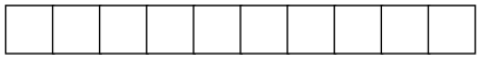

Figura 13. Matriz de PuntoDinamico vaciá.

Al iniciar nuestra aplicación la matriz ya inicia con 2 (dos) elementos almacenados que son los puntos de los extremos de la linea:

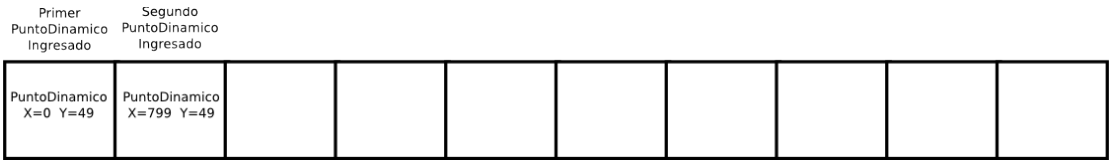

Figura 13. Matriz de PuntoDinamico con contenido.

A través de un ciclo **for** vamos a ir recorriendo esta matriz e ir leyendo uno por uno de sus elementos para poder dibujándolos en pantalla:

```
color colorTrazo = color(#0f5a62);
color rellenoTrazo = color(#fef9d4);

…

void dibujar() {
    for(int i=1;i<puntosDinamicos.length;i++) {
      if(puntosDinamicos[i] != null) {
        float x1 = puntosDinamicos[i-1].x;
        float y1 = puntosDinamicos[i-1].y;
        float x2 = puntosDinamicos[i].x;
        float y2 = puntosDinamicos[i].y;
        float radio = puntosDinamicos[i-1].radio;
        stroke(colorTrazo);
        fill(rellenoTrazo);
        line( x1, y1, x2,  y2);
        ellipse( x1, y1, radio, radio);
        ellipse( x2,  y2,  radio,  radio);
      }
    }
  }

…
```

De este modo nuestra linea quedaría de la siguiente manera (los números que se ven dentro de los círculos son solo ilustrativos para que el lector comprenda que que posición de la matriz se encuentra cada uno):

Figura 14. Linea dinámica con puntos.

Hasta este momento no se ha producido ninguna auto-organización puesto que los “PuntosDinamicos” que van ingresando son en orden creciente según la abscisa, esto quiere decir que el primer punto (circulo con numero 0) se coloco en el pixel 0 y el segundo (circulo con numero 1) en el 799. Al estar colocados en este orden cuando se recorre con el ciclo **for** la linea se forma de manera correcta y no pierde su figura de linea continua, el problema surge cuando los puntos ingresados no son de manera creciente.

Un ejemplo concreto puede ser cuando el usuario comienza a quebrar la linea entre los puntos **0 y 1**:

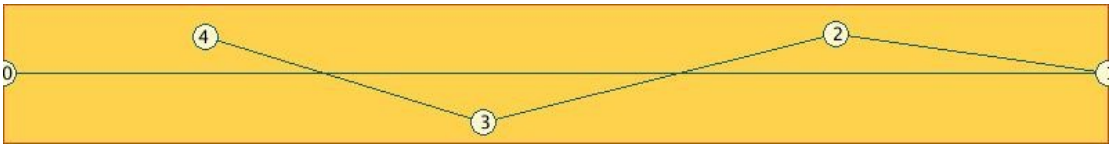

Figura 15. PuntosDinamicos almacenados.

lo que esta sucediendo en este caso es que el usuario ingreso puntos con el objetivo de quebrar la linea pero en vez de que eso suceda el algoritmo lo que esta haciendo es ir dibujando los puntos según el orden en que fueron ingresados en la matriz y NO en el orden creciente según el eje X.

Primer PuntoDinamico Ingresado	Segundo PuntoDinamico Ingresado	Tercero PuntoDinamico Ingresado	Cuarto PuntoDinamico Ingresado	Quinto PuntoDinamico Ingresado					
PuntoDinamico X=0 Y=49	PuntoDinamico X=799 Y=49	PuntoDinamico X=602 Y=21	PuntoDinamico X=348 Y=84	PuntoDinamico X=146 Y=23					

Figura 16. Matriz de PuntosDinamicos almacenados.

Aquí es donde entra en juego el “auto-ordenamiento”, en nuestro caso vamos a utilizar un algoritmo llamado “Ordenación de Bubuja o Bubble Sort” lo vamos a utilizar para ordenar de menor a mayor, según las abscisa, los elementos de la matriz y de este modo conseguir quebrar la linea correctamente.

El siguiente algoritmo es quien se encarga de ordenar los elementos de menor a mayor:

```
int[] ReacomodaNodosSegunEjeX (PuntoDinamico[] nodos_) {
  int cant = 0;
  for( int i=0 ; i<nodos_.length ; i++ ) {
    if(nodos_[i] != null) {
      cant++;
    }
  }

  float posiciones[] = new float[ cant];
  int indices[] = new int[cant];

  for( int i=0 ; i<cant ; i++ ) {
    posiciones[i] = nodos_[i].x;
    indices[i] = i;
  }

  for( int i=0 ; i<cant-1 ; i++ ) {
    for( int j=i+1 ; j<cant ; j++ ) {
      if( posiciones[j]<posiciones[i] ) {

        float auxTam = posiciones[i];
        posiciones[i] = posiciones[j];
        posiciones[j] = auxTam;

        int auxInd = indices[i];
        indices[i] = indices[j];
        indices[j] = auxInd;
      }
    }
  }
  return indices;
```

```
}
```

El método anterior recibe como parámetro una matriz de PuntosDinamicos y nos devuelve una matriz de tipo entera que contiene las posiciones de nuestro vector ordenadas de menor a mayor, para poder llevar a cabo tal proceso se va revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en orden equivocado. Es necesario revisar varias veces la lista hasta que no se necesiten mas intercambios, lo cual significa que la lista esta ordenada.

La implementación de la función anterior se lleva a cabo de la siguiente manera, teniendo que modificar la función **dibujar(),** analizada anteriormente:

```
int[] xIndex;
…
void agregarPunto(float x_,float y_) {
    int id = buscarLugarVacio();
    if(id > -1) {
      puntosDinamicos[id] = new PuntoDinamico(x_,y_);
      xIndex = ReacomodaNodosSegunEjeX(puntosDinamicos);
    }
  }


…


void dibujar() {
    for(int i=1;i<puntosDinamicos.length;i++) {
      if(puntosDinamicos[i] != null) {
        float x1 = puntosDinamicos[xIndex[i-1]].x;
        float y1 = puntosDinamicos[xIndex[i-1]].y;
        float x2 = puntosDinamicos[xIndex[i]].x;
        float y2 = puntosDinamicos[xIndex[i]].y;
        float radio = puntosDinamicos[i-1].radio;
        stroke(colorTrazo);
        fill(rellenoTrazo);
        line( x1, y1, x2,  y2);
        ellipse( x1, y1, radio, radio);
        ellipse( x2,  y2,  radio,  radio);
      }
    }
  }
```

En el codigo anterior puede verse como se crea la matriz **xIndex[],** quien va a ir almacenando los indices ordenados (cada vez que se ingresa un nuevo punto) que son retornados por la función **ReacomodaNodosSegunEjeX** , a la hora de dibujar los puntos en vez de tomar la posicion con **puntosDinamicos[i-1].x** y **puntosDinamicos[i].x** como se hacia en la funcion **dibujar()** mostrada anteriormente ahora se utiliza **puntosDinamicos[xIndex[i-1]].x** y **puntosDinamicos[xIndex[i]].x** , esto significa que cuando obtiene el primer elementos de puntosDinamicos lo hace por medio del primer elemento de **xIndex[]** que es quien tiene los indices ordenados.

En la siguiente imagen puede verse la matriz puntosDinamicos inicial y como queda una vez ordenada:

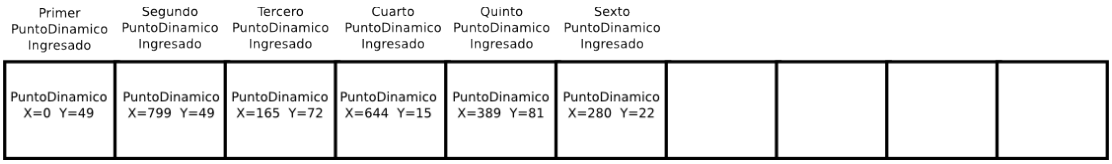

Figura 17. Matriz de PuntosDinamicos sin ordenar.

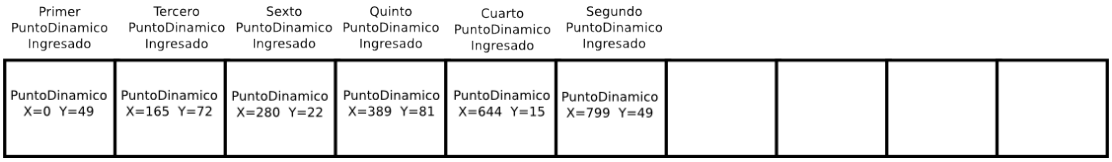

Figura 18. Matriz de PuntosDinamicos ordenada.

Y por ultimo la imagen de cómo quedaría la linea quebrada con los nodos ordenados:

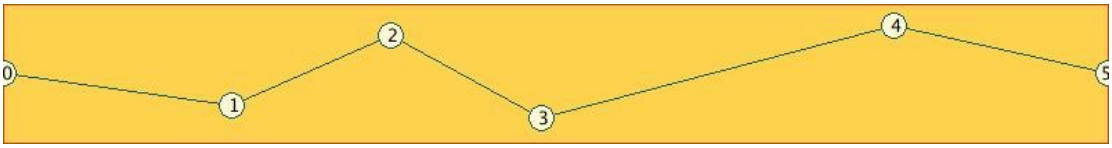

Figura 19. LineaDinamica quebrada correctamente.

3.3.3. Panel Edición

El objeto “**PanelEdicion**” nos permite dibujar, con el movimiento de nuestros dedos, grafías musicales sobre su superficie y a su vez se encarga de analizarlas, reconocerlas y catalogarlas según el tipo de grafía musical:

- Alturas puntuales.
- Diseño melódico.
- Acorde.
- Arpegios.
- Duración.
- Glissando.
- Trémolo.
- Trino.

El gesto correspondiente a cada grafía ya ha sido desarrollado y explicado en el documento “Diseño de interface para el desarrollo de una pantalla sensible al tacto con aplicación musical” del autor Emiliano Causa, por lo tanto, no se abordaran de manera detallada los mismos.

Este objeto se inicia y dibuja siguiendo la misma lógica y estructura algorítmica que los anteriormente nombrados, pertenecientes a la interfaz, por lo tanto pasaremos a explicar la parte de la actualización del mismo donde entran en juego cuatro objetos muy importantes que son “Empaquetador”, “AnalisisGestual”, “InterpreteGestual” y “EsteticaGrafias” a quienes les dedicaremos un apartado posteriormente.

El siguiente código es quien se encarga de la actualización del objeto:

```
…

void actualizar(NuevoMediador pantalla) {
    if(pantalla.cantPunteros > 0) {
      for(int i=0;i<pantalla.cantPunteros;i++) {
        Puntero este = pantalla.devolverPunteroNum(i);
        if(este != null) {
          if((punteroDentroDePanelEdicion(este)) && (!
punteroFueraDeBotones(este))) {
            activoRegion = true;
            empaquetador.empaquetaGestos(pantalla,este);
            empaquetador.activar(true);
          }
          else {
            activoRegion = false;
            empaquetador.actualizar();
            empaquetador.activar(false);
          }
        }
      }
    }
    else {
      activoRegion = false;
      empaquetador.actualizar();
      empaquetador.activar(false);
    }
    cambioEstado(activoRegion);
    analizarPanel();
  }

…
```

Como puede verse el método **actualizar** recibe como parámetro a un objeto **NuevoMediador** , al igual que el resto de los paneles consulta si un dedo esta presionando su región y también que no este tocando a los botones, esto se lleva a cabo en la siguiente linea **! punteroFueraDeBotones(este),** esta ultima condición se utiliza puesto que si el usuario presiona alguno de los botones también estaría presionando parte de la región del panel donde se dibujan grafías y lo que produciría seria que ingrese alguna grafía por debajo de los botones.

Si el dedo esta habilitado para dibujar grafías se activa el siguiente código:

```
  activoRegion = true;
  empaquetador.empaquetaGestos(pantalla,este);
  empaquetador.activar(true);
```

aquí se activa la región para que cambie su color para informarnos que estamos sobre su área correctamente y se ejecutan dos funciones del objeto "**Empaquetador**" que son **empaquetador.empaquetarGesto(pantalla, este)** y **empaquetador.activar(true)**, el primero se encarga de pasar como parámetro la instancia del objeto NuevoMediador y la instancia del objeto Puntero que esta en su región y el segundo activa al empaquetador. Estos métodos se utilizan para ir guardando todos los movimientos del dedo para su posterior análisis.

Cuando el dedo sale de la región se ejecuta la función **empaquetador.activar(false)** quien detiene el empaquetamiento y lo actualiza con **empaquetador.actualizar()** esto prepara al objeto para poder recibir y empaquetar otro dedo.

Paso seguido es activar el método analisisPanel(), este se ejecuta cuando dejan de existir dedos en la región por determinado umbral de tiempo y analiza –por decirlo de algún modo- lo sucedido en el área de dibujos grafías con el siguiente código:

```
void analizarPanel() {
    if(empaquetador.finalizoEmpaquetamiento()) {
      analisisGestual.analizar(empaquetador);
      interpreteGestual.interpretar(analisisGestual);
      if(interpreteGestual.existeMelodia) {

        esteticaGrafias.actualizar(interpreteGestual);
      }
      empaquetador.reset();
    }
  }
```

El código anterior es muy simple de entender puesto que su sintaxis esta escrita literalmente, por lo que cuando el empaquetador termina de empaquetar los dedos se activa el objeto análisis gestual quien analiza los dedos empaquetados, luego el interprete gestual interpreta dicho análisis y si encuentra en dicha interpretación la existencia de una melodía se lo informa al objeto estética grafías para que las dibuje, por ultimo se resetea el objeto empaquetador para poder volver a guardar nuevos dedos.

3.4. Empaquetador

El objeto ”**Empaquetador**” es el encargado de ir almacenando temporalmente los dedos ingresados en la pantalla. Registrado su evolución en el tiempo cual si fueran punteros y clasifica los dedos en estáticos y móviles.

Para poder llevar a cabo dichos procesos cuenta con una *MEF* (maquina de estados finitos) internamente quien modifica el comportamiento de este objeto según su estado:

- Espera.
- Procesando.
- Finalizado.

El siguiente método es quien se encarga de manejar los estados:

```
DedoGenerico[] dedoGenerico;
  int cantDedosGenerico = 100;
  int distanciaParaMovimiento = 30;


 boolean seActiva;


  int cuenta;
  int limite = 50;
  int cont = 0;
```

```
  int ESPERA = 1;
  int PROCESANDO = 2;
  int FINALIZADO = 3;
  int estado = ESPERA;

…

void analizarEstados() {
    if((estado != FINALIZADO) || (estado != ESPERA)) {
      agregoDedos();
      actualizarDedos();
    }
    if(estado == ESPERA) {
      if(seActiva) {
        estado = PROCESANDO;
      }
    }
    else if(estado == PROCESANDO) {
      if(!seActiva) {
        cuenta--;
        if(cuenta <= 0) {
          estado = FINALIZADO;
        }
      }
      else {
        cuenta = limite;
      }
    }
  }

…
```

Como puede verse al iniciar el código se declaran tres variables **ESPERA, PROCESANDO** y **FINALIZADO** que son los valores que ir almacenando la variable **estado**. Esta ultima inicia con el valor ESPERA, es decir que esta escuchando al objeto “PanelEdicion” haber cuando le informa que hay un dedo en su área.

```
…

if(estado == ESPERA) {
  if(seActiva) {
    estado = PROCESANDO;
  }
}
…
```

Cuando la variable **seActiva** para a tener el valor **TRUE** quiere decir que hay un dedo y **estado** pasa a tener el valor **PROCESANDO** , dada esta situación mientras exista un dedo en la pantalla la variable **cuenta** se setea con el valor almacenado en **limite** que es 50, esto se hace para que cuando un dedo se levanta y **seActiva** pasa a ser **FALSE** la variable **cuenta** empieza a restar sus valores hasta llegar a cero y cuando esto se cumple **estado** pasa a tener el valor **FINALIZADO**.
En otras palabras, la variable **cuenta** es el tiempo que existe para que pueda aparecer otro dedo en pantalla y ser parte de un mismo gesto.

```
    …
```

```
   else if(estado == PROCESANDO) {
      if(!seActiva) {
        cuenta--;
        if(cuenta <= 0) {
          estado = FINALIZADO;
        }
      }
      else {
        cuenta = limite;
      }
    }

   …
```

Por ultimo, cuando la variable estado no esta a la espera ni finalizada significa que aun esta en proceso, esto se corrobora con la linea **(estado != FINALIZADO) || (estado != ESPERA)** , si se encuentra en este estado se llevan a cabo los métodos **agregoDedos() , actualizarDedos()** como puede verse a continuación:

```
…

if((estado != FINALIZADO) || (estado != ESPERA)) {
      agregoDedos();
      actualizarDedos();
}
…
```

La función **agregoDedos()** se encarga de crear un objeto **DedoGenerico** cuando el dedo ingresa a la pantalla con su posicion actual en x e y, este es almacenado en una matriz de **DedoGenerico[]**, ahora bien, si el dedo comienza a moverse superando el umbral marcado con la variable **distanciaParaMavimiento** entra en juego la función **actualizarDedos()** quien comienza a actualizar al objeto **DedoGenerico** almacenando todas las posiciones por la que pasa, de este modo podemos diferenciar dos tipos de dedos: los estáticos y los en movimiento. Esto nos sirve para poder consultar a futuro que dedos generaron trayectos y cuales permanecieron en el lugar para poder analizar con el objeto "**AnalisisGestual**" que grafía hizo cada uno.

Llegado a este punto todos los trayectos graficados por los dedos son dibujados en pantalla, aun sin importar si corresponden a grafías musicales o no, esto sirve para que el usuario vea, mientras mueve los dedos sobre la pantalla, el gesto que esta creando. El encargado de realizar esta acción es la siguiente función:

```
…

void dibujo() {
    stroke(0)
    fill(255);
    for(int i=0;i<dedoGenerico.length;i++) {
      if(dedoGenerico[i] != null) {
         if(dedoGenerico[i].trayectoria.length > 1) {
            for(int j=1;j<dedoGenerico[i].trayectoria.length;j++) {
              line(dedoGenerico[i].trayectoria[j-1].x,
dedoGenerico[i].trayectoria[j-1].y,
dedoGenerico[i].trayectoria[j].x,dedoGenerico[i].trayectoria[j].y);
            }
          }
          else {
```

```
ellipse(dedoGenerico[i].trayectoria[0].x,dedoGenerico[i].trayectoria[0].y,
tamanio,tamanio);
          }
      }
    }
 }

...
```

El algoritmo anterior recorre todos los datos almacenados en **dedoGenerico[]** y consulta si alguno de esos dedos tiene trayectoria con **if(dedoGenerico[i].trayectoria.lenght > 1)** si esto se cumple se dibuja una linea continua, sino se cumple se dibuja solo un circulo en una posición determinada.

3.5. Analisis Gestual

El objeto "**AnalisisGestual**" es el encargado de analizar los diferentes tipos de movimientos que puede generar cada dedo en el desarrollo de su gesto, este análisis incluye la generación de descripciones respecto a la rectitud o zigzageos de los trazos, la dirección, la curvatura, así como sus posiciones relativas en función de los demás dedos. En otras palabras, se encarga de traducir datos cartesianos en parámetros perceptivos.

La principal función de este objeto es la de analizar los datos almacenados por el objeto "Empaquetador":

```
  AnalisisMovimiento analisis;
  DedoEstatico[] dedoEstatico;
  DedoMovil[] dedoMovil;

  Ordenador ooo = new Ordenador();

…

void analizar(Empaquetador empaquetador_) {
    empaquetador = empaquetador_;
    cantDatos = empaquetador.CantidadDedosEmpaquetados();

    cantDedosQuietos = CantidadDedosQuietos();
    cantDedosMoviles = CantidadDedosMoviles();
    dedoEstatico = new DedoEstatico[cantDedosQuietos];
    dedoMovil = new DedoMovil[cantDedosMoviles];

    …

  }
…
```

Al iniciar el código se declaran dos matrices de objetos DedoEstatico[] y DedoMovil[] estos irán almacenando los dedos quietos y los móviles respectivamente, cada uno de estos objetos contendrá información diferente:

1. DedoEstatico:
    a. El identificador del dedo.
    b. La posición en X e Y.
    c. La posición respecto a un yIndex y xIndex.
2. DedoMovil:
    a. El tipo de orientación.
    b. El tipo de variación.
    c. El identificador del dedo.
    d. La trayectoria del mismo.
    e. La distancia total que mide el trazo.
    f. La posición en X e Y de los puntos inicial y final de la linea o trazo.
    g. Los xIndex y yIndex de los puntos inicial y final de la linea o trazo.

Dos pasos muy importantes se llevan a cabo este objeto, el primero como puede verse tanto para los dedos estáticos como para los móviles, es la obtención de sus **xIndex** e **yIndex** estos son importante para tener el orden en que están visualmente ubicados en pantalla sin importar el orden en que fueron ingresado. En otras palabras, **yIndex** y **xIndex** son las posiciones de los puntos ordenadas con el algoritmo de “burbuja” explicado mas arriba.

La siguiente imagen es un ejemplo de cómo se ordenan los puntos, ya sean dedos móviles (lineas) o dedos estáticos (círculos).

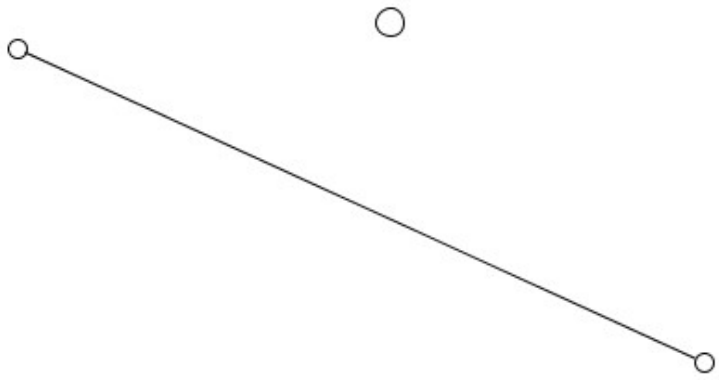

Figura 20. Puntos a ser ordenados.

En este caso se ha creado primero el dedo móvil y luego sobre entre sus extremos un dedo estático, planteado de este modo se almacenaría el dedo móvil y luego el estático pero perceptivamente esto no es así, los datos sin ser ordenados se presentan de la siguiente manera:

```
lineaInicial[0]->( X=84.0 | Y=124.0 ) ind.X=0 ind.Y=0
lineaFinal[0]->( X=454.0 | Y=293.0 ) ind.X=0 ind.Y=0
punto[0]->( X=285.0 | Y=110.0 ) ind.X=0 ind.Y=0
```

Los datos ordenados quedarían de la siguiente manera:

```
lineaInicial[0]->( X=84.0 | Y=124.0 ) ind.X=0 ind.Y=1
lineaFinal[0]->( X=454.0 | Y=293.0 ) ind.X=2 ind.Y=2

punto[0]->( X=285.0 | Y=110.0 ) ind.X=1 ind.Y=0
```

Al tenerlos ordenados de esta manera se nos es mas fácil poder armar la partitura musical ya que sabemos el orden en X e Y en el que se encuentran.

El segundo paso importante, que solo compete a los dedos móviles, es la obtención de la orientación y variación de los trazos, esto se lleva a cabo con un objeto llamado "**AnalisisMovimiento**" el cual ha sido explicado en el documento "Diseño de interface para el desarrollo de una pantalla sensible al tacto con aplicación musical" del autor Emiliano Causa.

3.6. Interprete Gestual

El objeto "**InterpreteGestual**" recibe la descripción hecha por el análisis gestual pero a diferencia de este hace un análisis integral del comportamiento en conjunto de todos los dedos en la configuración de un "gesto", se encarga de analizar cuando empieza y termina un "gesto", así como de revisar si estos cumplen con "una buena forma".

Para poder llevar a cabo el análisis integral se implementa un algoritmo desarrollado por el Ing. Emiliano Causa, en el documento "Diseño de interface para el desarrollo de una pantalla sensible al tacto con aplicación musical", durante el cual se evalúan las diferentes condiciones que permiten distinguir a cada uno de los gestos.

A continuación se explicara como se incorporan los gestos, una vez reconocidos, al objeto "**Partitura**" sin entrar en detalles sobre el funcionamiento del algoritmo en sí, puesto que ya fue explicado en detalle en el otro documento anteriormente nombrado.

Una aclaración importante es que en todas estas funciones las grafías musicales que se almacenan en el objeto "Partitura" están siendo escaladas, así que cuando figure la función **devolverAlturasEscaladas()** ya sabemos a que hace referencia.

Nota Puntual:

```
…
void NotaPuntual() {
    for (int i=0;i<cantidadDedosQuietos;i++) {
      float x = receptor.dedoEstatico[i].x;
      int y = devolverAlturasEscaladas(receptor.dedoEstatico[i].y);
      partitura.agregarNota(x,y);
    }
}

…
```

Para ingresar las notas puntales solo basta con recorrer todos los dedos quietos que se encuentran en los gestos y obtener su posición en **X** y su valor en **Y**, a este ultimo se lo obtiene con la función **devolverAlturasEscaladas()** quien nos convierte el valor del pixel en una escala musical (explicado en los apartados anteriores). Por ultimo estos valores se ingresan en **partitura.agregarNota(x,y)**.

Duracion o Trino:

```
…

void Duracion_o_Trino(String queTipo_) {
    float x1 = 0;
    float y1 = 0;
    float x2 = 0;
    float y2 = 0;
    int duracion_o_trino = 0;

    for (int i=0;i<cantidadDedosQuietos;i++) {
      x1 = receptor.dedoEstatico[i].x;
      y1 = receptor.dedoEstatico[i].y;
    }
    for(int i=0;i<cantidadDedosMovil;i++) {
      x2 = receptor.dedoMovil[i].xFinal;
      y2 = receptor.dedoMovil[i].yFinal;
    }

    if(queTipo_.equals("DURACION")) {
      duracion_o_trino = (int)dist(x1,y1,x2,y2);
 partitura.agregarDuracion(x1,devolverAlturasEscaladas(y2),duracion_o_trino)
;
    }

    else if(queTipo_.equals("TRINO")) {
      duracion_o_trino = (int)dist(x1,y1,x2,y2);
       partitura.agregarTrino(x1,devolverAlturasEscaladas(y1),duracion_o_trin
o);
    }
  }

…
```

Esta función recibe como parámetro el tipo de gesto que se desea almacenar ya sea duración o trino, para cualquiera de ambos se necesita obtener el x e y del dedo quieto y el x e y final del dedo móvil,

por esta razón se recorren con un **ciclo for** todos los dedos quietos, obteniendo sus valores con **receptor.dedoEstatico[i].x** y **receptor.dedoEstatico[i].y** los cuales se almacenan en **x1, y1** respectivamente, luego se recorrer todos los dedos moviles y se obtienen sus valores de x e y finales con **receptor.dedoMovil[i].xFinal** y **receptor.dedoMovil [i].yFinal** almacenándolos en **x2, y2** respectivamente.

Una vez que obtenemos estos datos **x1, y1, x2, y2** procedemos a hacer los procesos necesarios para la duración en donde calculamos la distancia entre los puntos x1 e y1 del dedo quieto y x2 e y2 del dedo móvil con **dist(x1,y1,x2,y2)** y lo almacenamos en **duracion_o_trino**, por ultimo lo agregamos a la partitura con **partitura.agregarDuracion(x1 , devolverAlturasEscaladas(y2) , duracion_o_trino).** El mismo proceso se repite para el trino a diferencia que a la partitura se agrega con **partitura.agregarTrino(…).**

Diseño Melódico:

```
…

void Melodia() {
    for(int i=0;i<cantidadDedosMovil;i++) {
      int cantidad = receptor.dedoMovil[i].recorrido.length;
      float xIni = receptor.dedoMovil[i].xInicial;
      float xFin = receptor.dedoMovil[i].xFinal;
      if(xIni < xFin) {
        int contadorNotas = 0;

        for(int j=0;j<cantidad;j++) {
          if(contadorNotas == 0) {
            partitura.agregarMelodia();
          }
          float x = receptor.dedoMovil[i].recorrido[j].posX;
          float y =
devolverAlturasEscaladas(receptor.dedoMovil[i].recorrido[j].posY);
          partitura.agregarTrayectoMelodia(j,x,y);
          contadorNotas++;
          if(contadorNotas == 6) {
            contadorNotas = 0;
          }
        }
      }
      else {
        int contadorNotas = 0;

        for(int j=cantidad-1;j>0;j--) {
          if(contadorNotas == 0) {
            partitura.agregarMelodia();
          }

          float x = receptor.dedoMovil[i].recorrido[j].posX;
          float y =
devolverAlturasEscaladas(receptor.dedoMovil[i].recorrido[j].posY);
          partitura.agregarTrayectoMelodia(j,x,y);
          contadorNotas++;
          if(contadorNotas == 6) {
            contadorNotas = 0;
          }
        }
      }
    }
  }
```

```
…
```

La función melodía recorre todas las posiciones x e y por las que paso el trayecto de un dedo móvil y se encarga de dos cosas, primero de analizar como recorrer todas las posiciones, esto quiere decir que si el trazo inicio de derecha a izquierda la función los procesara de izquierda a derecha con el fin de que en la partitura queden guardadas en un orden temporal coherente. Lo segundo que se encarga es de corroborar cuantas notas integra el diseño melódico, si contiene mas de 6 notas crea un nuevo diseño melódico para almacenar las restantes.

Para generar la partitura primero se debe agregar la melodía con **partitura.agregarMelodia()** y luego se guardan todas los puntos (notas) que integra con **partitura.agregarTrayectoMelodia(j,x,y)** donde **j** es en identificador, **x** e **y** las posiciones.

La partitura se tiene que generar de este modo puesto que esa es la cantidad de notas que se utilizan para construir los Pitch Class Sets, es decir, los conjuntos de notas coherentes (según este sistema) máximo para construir las agrupaciones de notas que conforman las melodías, acordes y arpegios.

Acorde:

```
…

void acorde() {
    int contadorNotas = 0;
    for (int i=0;i<cantidadDedosQuietos;i++) {
      float x = receptor.dedoEstatico[indiceX[i]].x;
      float y =
devolverAlturasEscaladas(receptor.dedoEstatico[indiceX[i]].y);
      if(contadorNotas == 0) {
        partitura.agregarAcorde();
      }
      partitura.agregarComponenteAcorde(i,x,y);
      contadorNotas++;
      if(contadorNotas == 6) {
        contadorNotas = 0;
      }
    }
  }

…
```

Al igual que la función de melodía, en el análisis de acorde también se debe tener en cuenta cuantas notas integra en su gesto, en este caso en relación a los dedos quietos. Para agregar el acorde a la partitura se utiliza **partitura.agregarAcorde()** y para incorporarle sus notas se utiliza **partitura.agregarComponenteAcorde(i,x,y)** donde **i** es el identificador, **x** e **y** las posiciones de las notas.

Arpegio:

```
…

void arpegio() {
    int contadorNotas = 0;
    for (int i=0;i<cantidadDedosQuietos;i++) {
      float x = receptor.dedoEstatico[indiceX[i]].x;
      float y =
```

```
devolverAlturasEscaladas(receptor.dedoEstatico[indiceX[i]].y);
      if(contadorNotas == 0) {
        partitura.agregarArpegio();
      }
      partitura.agregarComponenteArpegio(i,x,y);
      contadorNotas++;
      if(contadorNotas == 6) {
        contadorNotas = 0;
      }
    }
  }

…
```

La función arpegio se encarga de recorrer los dedos quietos que integran su grafía musical y, al igual que la melodía y el acorde, si integra mas de 6 notas crea uno nuevo. Para agregar el arpegio a la partitura se utiliza **partitura.agregarArpegio()** y para incorporarle sus notas se utiliza **partitura.agregarComponenteArpegio(i,x,y)** donde **i** es el identificador, **x** e **y** las posiciones de la nota.

Glissando:

```
…

void glissando() {
    float x1 = 0;
    float y1 = 0;
    float x2 = 0;
    float y2 = 0;
    for (int i=0;i<cantidadDedosQuietos;i++) {
      x1 = receptor.dedoEstatico[i].x;
      y1 = devolverAlturasEscaladas(receptor.dedoEstatico[i].y);
    }
    for(int i=0;i<cantidadDedosMovil;i++) {
      x2 = receptor.dedoMovil[i].xFinal;
      y2 = devolverAlturasEscaladas(receptor.dedoMovil[i].yFinal);
    }
    partitura.agregarGlissando(x1,y1,x2,y2);
  }

…
```

La función glissando se encarga de obtener por un lado la posición en **x** e **y** del dedo quieto y por otro la última posición en **x** e **y** del dedo móvil, teniendo estos datos se ingresa a la partitura con **partitura.agregarGlissando(x1,y1,x2,y2)** donde **x1** e **y1** son los datos del dedo quieto y **x2** e **y2** los del dedo móvil.

Tremolo:

```
…

void tremolo() {
    float tiempo = receptor.dedoEstatico[indiceX[0]].x;
    float alturaInicial =
```

```
devolverAlturasEscaladas(receptor.dedoEstatico[indiceX[0]].y);
    float alturaFinal =
devolverAlturasEscaladas(receptor.dedoEstatico[indiceX[1]].y);
    partitura.agregarTremolo(tiempo, alturaInicial,alturaFinal);
}

…
```

La función trémolo o se encarga de obtener el tiempo en el que inicia **receptor.dedoEstatico[indiceX[0]].x** que es almacenada en la variable **tiempo**, luego la altura inicial con **devolverAlturasEscaladas(receptor.dedoEstatico[indiceX[0]].y)** que es almacenada en la variable **alturaInicial** y por ultimo la altura final con **devolverAlturasEscaladas(receptor.dedoEstatico[indiceX[1]].y)** y es almacenada en **alturaFinal**, teniendo estos datos se incorpora en tremolo a la partitura con **partitura.agregarTremelo(tiempo,alturaInicial,alturaFinal)**.

3.7. Partitura y Ordenador Secuencial

El objeto "**Partitura**" se encarga de recibir y almacenar los elementos musicales interpretados a partir de los gestos, funciona como si fuera un "recipiente" contenedor de gestos, esto quiere decir que a medida que los recibe (a los elementos musicales) del "**interpreteGestual**" los va almacenando: todas las notas por un lado, los diseños melódicos por otro y así sucesivamente.

De este modo tenemos todos los elementos guardados por separado, por lo que tenemos que hacer uso del objeto "**OrdenadorSecuencial**" quien se encarga, como su nombre lo indica, de ordenarlos secuencialmente, producto de este ordenamiento surge el xml que recibirá el modulo de Pure Data.

El siguiente método se encuentra en el objeto "**Partitura**" y es quien se encarga de agregar al objeto "**OrdenadorSecuencial**" todos los elementos a ser ordenados:

```
…

void ordenarSecuencia() {
    ordenador.vaciar();
    ordenador.agregar("NOTA",notaPuntual);
    ordenador.agregar("TRINO",trino);
    ordenador.agregar("MELODIA",melodia);
    ordenador.agregar("ACORDE",acorde);
    ordenador.agregar("ARPEGIO",arpegio);
    ordenador.agregar("GLISSANDO",glissando);
    ordenador.agregar("TREMOLO",tremolo);
    ordenador.ordenar();

 }

…
```

El algoritmo anterior se encarga de:

- Vaciar o limpiar los elementos que hayan quedado almacenados de la partitura anterior.
- Agregar los elementos musicales, donde se le informa el tipo de elemento que es y los elementos propiamente dichos.
- Una vez que se agregan todos los elementos se encarga de ordenarlos.

Para poder ordenarlos el objeto "**OrdenadorSecuencial**" cuenta con los siguientes métodos:

```
…

void ordenar() {
    ordenEnX = ordenarVector( x );
    for ( int i=0 ; i<cant ; i++ ) {
      indiceX[ ordenEnX[i] ] = i;
    }
    ordenEnY = ordenarVector( y );
    for ( int i=0 ; i<cant ; i++ ) {
      indiceY[ ordenEnY[i] ] = i;
    }
  }

…

int[] ordenarVector( float arreglo[] ) {
    float valor[] = new float[ cant ];
    int cual[] = new int[ cant ];

    for ( int i=0 ; i<cant ; i++ ) {
      cual[i] = i;
      valor[i] = arreglo[i];
    }

    for ( int i=0 ; i<cant-1 ; i++ ) {
      for ( int j=i+1 ; j<cant ; j++ ) {
        if ( valor[j] < valor[i] ) {

          float aux = valor[i];
          valor[i] = valor[j];
          valor[j] = aux;

          int auxId = cual[i];
          cual[i] = cual[j];
          cual[j] = auxId;
        }
      }
    }
    return cual;
  }

…
```

El código anterior contiene el “Ordenamiento de Burbuja” (explicado en los capítulos anteriores) quien se encarga de ordenar en la línea de tiempo los elementos musicales, al tenerlos ordenados ya podemos escribir el código XML que será el mediador entre Processing y PureData.

La estructura del archivo xml ya ha sido explicada en el documento “Diseño de interface para el desarrollo de una pantalla sensible al tacto con aplicación musical” del Ing. Emiliano Causa.

4. Interpretación Musical de Datos (Conexión Superficie – Pure Data)

4.1. Estructura del Programa desarrollado en Pure Data

La aplicación en Pure Data se divide en cinco tareas principales. Algunas de ellas se inician en forma paralela a otros procesos mientras que otros son el resultado de un proceso de pasos en cadena. Estas tareas principales de la estructura del programa son:

- La generación del campo armónico a utilizar
- El ingreso y lectura del archivo XML
- La secuenciación de eventos en los objetos “coll”
- La substitución de datos musicales ingresados de la superficie con los generados en el campo armónico
- La reproducción MIDI

Además, parte de estos pasos son procesos independientes a la reproducción de cada secuencia nueva proveniente de la mesa. La generación del campo armónico se genera cada vez que se inicia la aplicación. Cualquier modificación que se quiera hacer al campo armónico debe hacerse ingresando al patch de Pure Data. Esto funciona así ya que en este estado inicial de la superficie, no se ofrecen posibilidades de control al usuario común.

4.2. Ingreso de Datos Musicales

El ingreso de los datos musicales se logra por medio de un archivo XML generado por *Processing* después del sensado y el análisis proveniente de la interface táctil. Para ellos se utilizo el objeto “Detox” en PD el mismo que es desarrollado como un external en la librería “jasch_lib” desarrollado por Jan Schacher. Dicho objeto permite leer el archivo XML como cualquier texto pero informa al usuario cuando se abren o se cierran las “llaves” del archivo. Los datos son después ruteados a una serie de abstracciones que se guardan en objetos “coll”. Esta parte del programa fue desarrollado por el Lic. Matías Romero Costas. Posteriormente fue complementado por la parte del programa que mapea los valores de la superficie a los valores de las alturas del campo armónico, esto será explicado mas adelante.

El XML cuenta con un formato propio establecido después de un estudio y desarrollo del mejor protocolo posible a seguir. En carácter general, cada gesto indica todas las notas musicales que lo componen y el tiempo de inicio y la duración de cada una de ellas. Estos datos están anidados en la etiqueta correspondiente a cada gesto musical según corresponda.

La estructura básica del archivo XML generado es la siguiente. (Los valores variables ingresados se encuentran en carácter de ejemplo)

```
<PICTO>
<TEMPO >
 <CANTIDAD> 30 </CANTIDAD>
  <MT>
            <P> 100 </P>
            <V> 60 </V>
  </MT>
  <MT>
            <P> 1000 </P>
            <V> 100 </V>
  </MT>
  <MT>
            <P> 5000 </P>
            <V> 40 </V>
            </MT>
</TEMPO >

<ACORDE ID='110'>
  <CANTIDAD> 3 </CANTIDAD>
  <NOTA>
    <T> 1000 </T>
    <A> 36 </A>
    <DIN> 80 </DIN>
    <D> 500 </D>
  </NOTA>
  <NOTA>
    <T> 1000 </T>
    <A> 40 </A>
    <DIN> 90 </DIN>
    <D> 500 </D>
  </NOTA>
  <NOTA>
    <T> 1000 </T>
    <A> 43 </A>
    <DIN> 100 </DIN>
    <D> 500 </D>
  </NOTA>
```

```
    <T> 1100 </T>
    <A> 48 </A>
    <DIN> 90 </DIN>
    <D> 1500 </D>
  </NOTA>
  <NOTA>
    <T> 1200 </T>
    <A> 52 </A>
    <DIN> 100 </DIN>
            <D> 500 </D>
  </NOTA>
  <NOTA>
    <T> 1300 </T>
    <A> 55 </A>
    <DIN> 110 </DIN>
            <D> 1000 </D>
  </NOTA>
  <NOTA>
    <T> 1400 </T>
    <A> 57 </A>
    <DIN> 120 </DIN>
            <D> 2000 </D>
  </NOTA>
</ARPEGIO>



<NOTA ID='111'>
    <T> 1200 </T>
    <A> 89 </A>
    <DIN> 70 </DIN>
    <D> 500 </D>
</NOTA>


<MELODIA ID='112'>
 <CANTIDAD> 5 </CANTIDAD>
```

```
</NOTA>
  <NOTA>
    <T> 3000 </T>
    <A> 64 </A>
    <DIN> 90 </DIN>
    <D> 500 </D>
  </NOTA>
  <NOTA>
    <T> 4000 </T>
    <A> 66 </A>
    <DIN> 90 </DIN>
    <D> 500 </D>
  </NOTA>
  <NOTA>
    <T> 5000 </T>
    <A> 67 </A>
    <DIN> 90 </DIN>
            <D> 500 </D>
  </NOTA>
  <NOTA>
    <T> 6000 </T>
    <A> 69 </A>
    <DIN> 90 </DIN>
            <D> 500 </D>
  </NOTA>
</MELODIA>


<TREMOLO ID='113'>
            <T> 1000 </T>
    <A> 77 </A>
            <AF> 90 </AF>
    <DIN> 80 </DIN>
            <D> 50 </D>
</TREMOLO>
```

```
<DIN> 60 </DIN>
    <D> 50 </D>
</GLISSANDO>

<TRINO ID='116'>
    <T> 500 </T>
    <A> 70 </A>
    <DIN> 120 </DIN>
    <D> 20 </D>
</TRINO>

</PICTO>
```

4.3. Reproductores de Gestos

Los gestos musicales implementados en la superficie táctil son: “acorde”, “trino”, “tremolo”, “melodía”, “nota”, “arpegio” y “glissando”. Todos ellos fueron implementados de manera que agrupe en una sola acción todos los parámetros que constituyen dicho gesto. Para ello, los reproductores arman en mensajes a manera de *buffers* o memorias temporales que alimentan los objetos “makenote” y “noteout” que arman mensajes midi y los mandan por las correspondientes salidas. Estos mensajes reciben los datos de los objetos “coll” que almacenan en ellos todas las acciones y las entregan una por una cuando reciben su índice correspondiente que equivale al momento en que deben darse.

4.3.1. Tremolo y Trino

Esencialmente, el programa de trino y del tremolo son el mismo en un noventa y nueve por ciento dado que el gesto funciona de la misma manera. Genera dos mensajes *buffer* que guardan los dos valores a intercalar a cierta densidad cronométrica establecida por el tempo y durante el tiempo total designado establecido por la duración del evento. La única diferencia se encuentra en que el reproductor de tremolo setea los dos mensajes con las dos alturas provenientes de la superficie, mientras que el reproductor de trino solamente recibe una altura y “trina” con un semitono arriba, por lo cual solo necesita la altura base a la cual se suma 1.

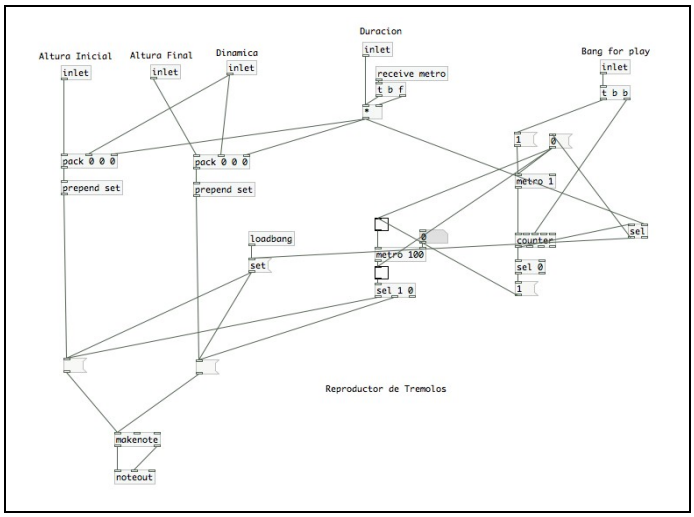

Figura 21. Reproductor de "Tremolos".

4.3.2. Acorde

El reproductor de acordes utiliza un solo mensaje como *buffer* y arma un mensaje con todas las alturas del acorde. El programa debe armar correctamente el mensaje intercalando las alturas y la velocidad de las notas para que el objeto "makenote" reproduzca el acorde adecuadamente. Para este reproductor así como también el de "arpegio", fue necesario modificar las abstracciones donde se encuentran los objetos "coll" para filtrar los distintos tiempo de inicio que entraban de la superficie táctil y establecer solo el primer tiempo como el mismo para todos. Esto se logró con la abstracción "once" desarrollada por Thomas Musil en Austria. Esta permite que solo el primer mensaje pase siendo una compuerta de mensajes.

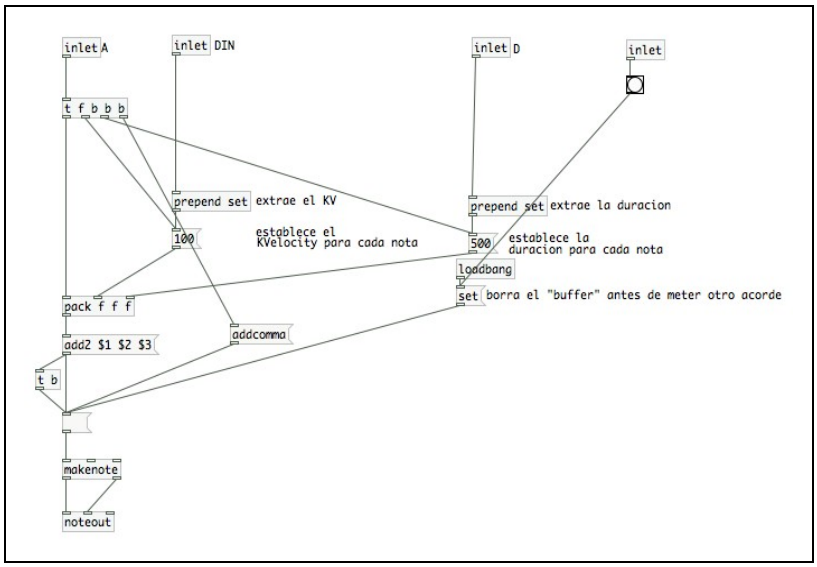

Figura 22. Reproductor de "Acordes".

4.3.3. Arpegios

La reproducción del arpegio, al igual que el del acorde, resulto con la misma modificación de las abstracciones donde se encuentra su correspondiente objeto “coll”. Además, el arpegio implico el uso del tiempo de inicio de la primera nota que lo constituye para después hacer un programa que los toque en sucesión. Para esto, se tuvo que utilizar 5 mensajes como memoria temporal donde iban a estar almacenadas las notas del arpegio (5 como máximo) que luego serian ejecutadas por un pequeño contador que los ejecutaría uno por uno en una sucesión rápida, característica fundamental del arpegio.

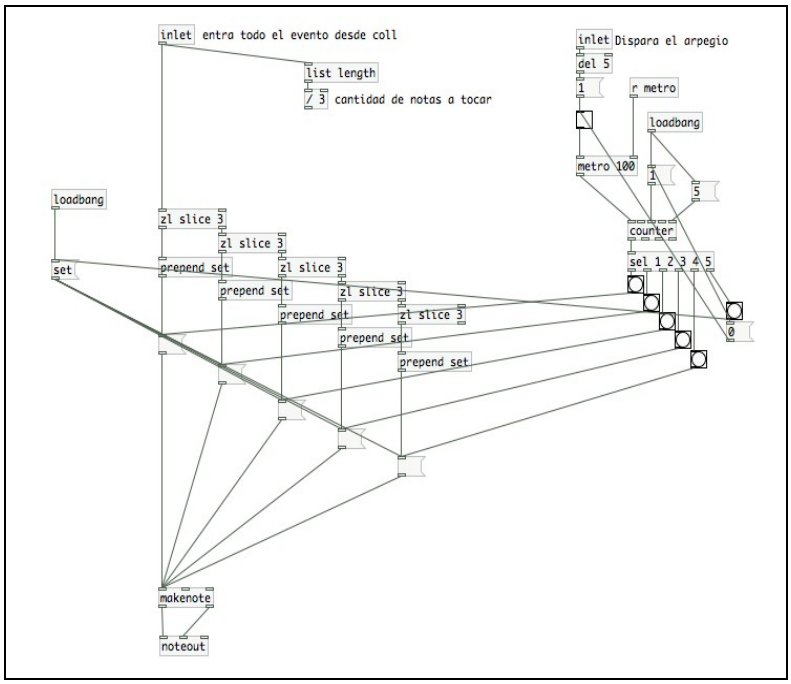

Figura 23. Reproductor de “Arpegios”.

4.3.4. Glissandos

El reproductor de glissandos es muy sencillo ya que siempre hace una interpolación lineal entre una nota y otra. Para ello se utiliza la nota de la que se parte y la nota a la que se llega y se interpola linealmente entre estas dos alturas utilizando el objeto “step”, este permite establecer también el tamaño del incremento.

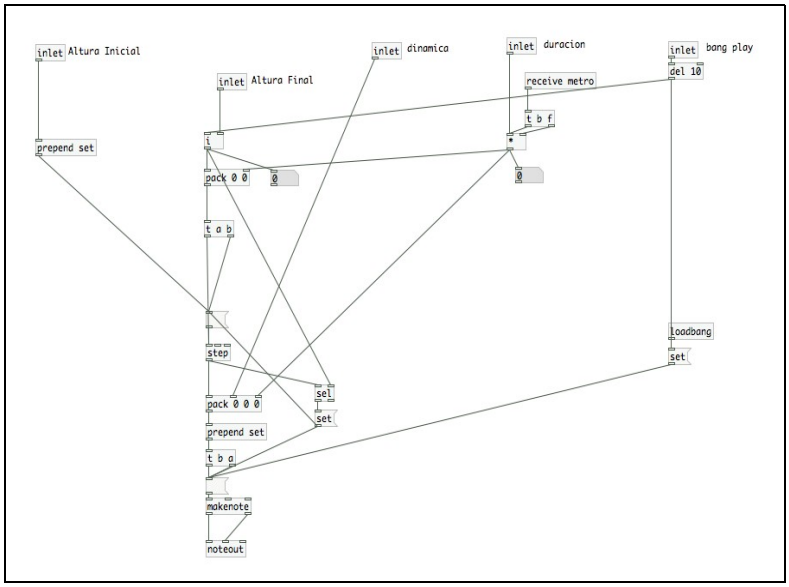

Figura 24. Reproductor de "Glissando".

4.3.5. Melodía y Notas

Los dos gestos "melodía" y "nota" utilizaran el mismo reproductor llamado "NotaPlayer" ya que sus respectivas abstracciones donde se encuentran los objetos "coll" correspondientes a ese gesto, entregan uno por uno el valor de la nota a reproducir. Permitiendo el uso de un reproductor de notas individuales ya que el gesto se puede dividir temporalmente sin ningún problema y cada nota puede durar lo que quiera.

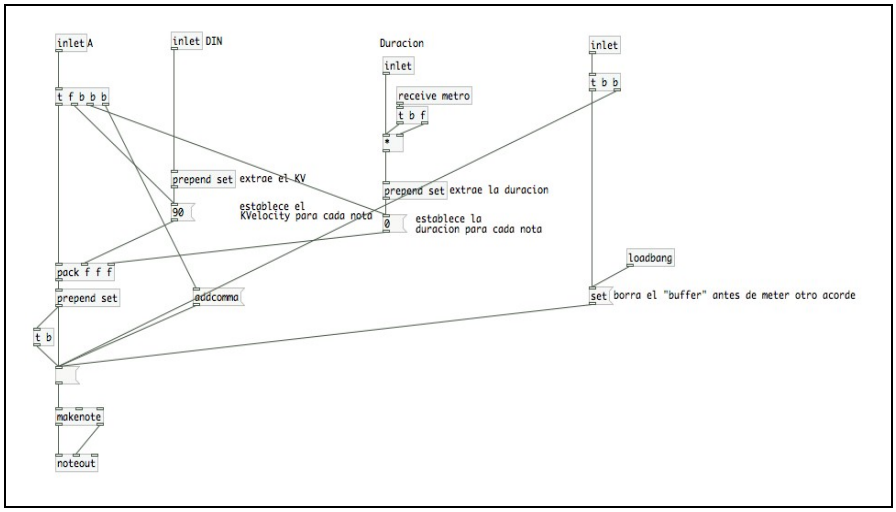

Figura 25. Reproductor de "Notas".

5. Estructuración de las alturas con PCSlib

5.1. Melodía y Notas

El campo armónico a utilizar se elabora utilizando un objeto de la librería PCSlib llamado “pcs_chain”. Dicho objeto elabora cadenas de *Pitch Class Sets* de una misma norma. Este esta parte del programa se encuentra en un subpatch y permite elegir un PCS de cinco o seis notas, a continuación el programa partirá el PCS en todas las posibles combinaciones de dos partes para luego el usuario elegir que partición ingresar a la cadena para comenzarla. El objeto entrega un listado con todas las posibles particiones a encadenar, la ultima opción siempre es la que mas se ajusta a la saturación de todo el total cromático.

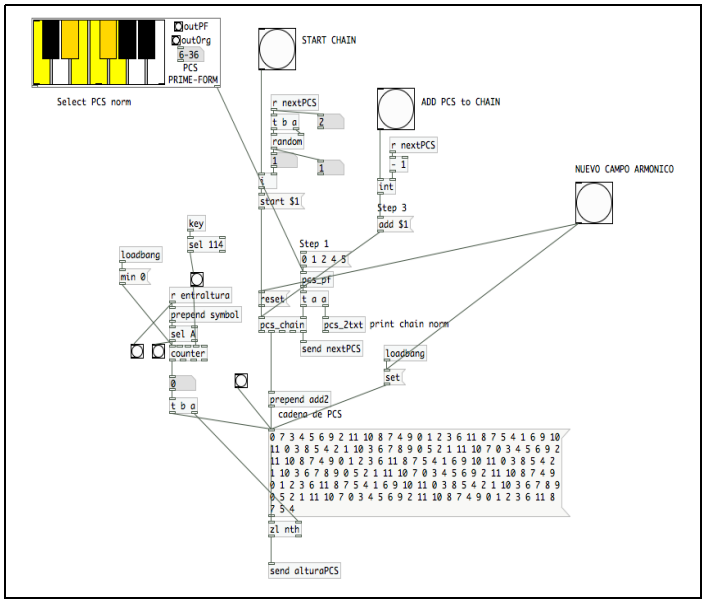

Figura 26. Creador del Campo Armonico.

5.2. Adaptación del campo armónico al gesto escrito

El campo armónico será la estructura fundamental para cualquier gesto escrito tenga un color definido de alturas. Para lograr esto, se debe comprometer la relación entre el gesto escrito, que entrega valores MIDI discretos, y las alturas que se acomodarán a las generadas en el campo armónico. Esto permite una coherencia musical de cualquier fragmento de música escrito en la superficie y permite mayor libertad al usuario para concentrarse en el gesto musical sin la preocupación de las alturas a sonar.

Para lograr esto, se tuvo que desarrollar varios algoritmos que permiten buscar la mejor acomodación de las alturas provenientes de campo armónico a los gestos escritos. Así evitamos la distorsión exagerada de lo que suena con lo que fue dibujado en la superficie. Un ejemplo claro de dicha distorsión exagerada seria la acomodación del dibujo a alturas que se encuentran muy lejos del registro y ámbito del gesto.

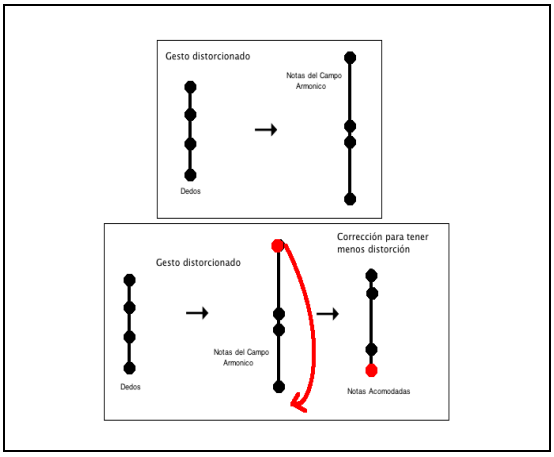

Figura 27. Esquema de funcionamiento del Algoritmo que trabaja con las permutaciones.

Por medio de distintos algoritmos logramos acomodar las alturas. Estas soluciones tuvieron que ser diferentes porque la lógica de los Pitch Class Sets se relaciona de manera particular con cada gesto. Esto quiere decir que el algoritmo de la ilustración anterior no va a funcionar para acomodar las alturas del gesto llamado “melodía”. Sin embargo, todas las soluciones particulares a este problema cuentan con una parte de programación que analiza la entrada de datos proveniente de la superficie. Este análisis ayudará entonces a comparar las alturas del gesto con las alturas del campo armónico. Para ello se ordenan los datos ingresante y extraen las alturas eliminando la relación de octava. Por otro lado, sí se guarda registro de a que octava corresponden las notas ingresante, por ultimo con ayuda del objeto “pcs_pf” se averigua que *set class* es. Toda la programación se logro gracias a las abstracciones incluidas en “Pd-extended” conocidas como “list-abs”. Estas abstracciones permiten la manipulación de listas, pilar fundamental en la programación algorítmica para el control de alturas.

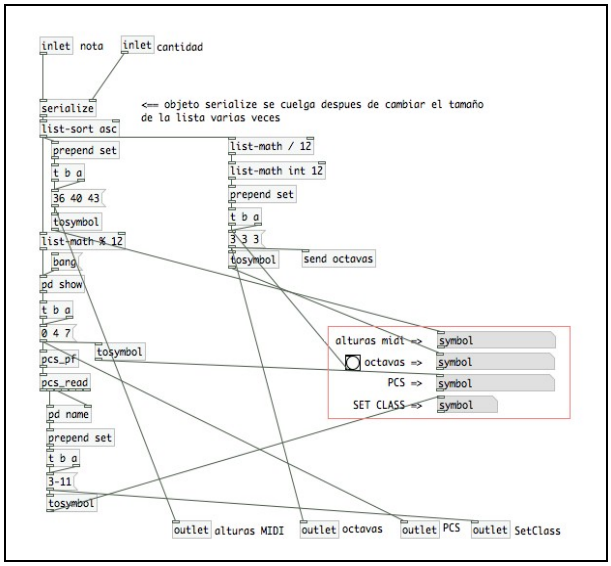

Figura 28. Programa que analiza las notas del gesto entrante.

5.2.1. Mapeo del gesto “acorde” y “arpegio” a los PCS

El algoritmo de adaptación de las alturas al gesto del “acorde” y “arpegio” es el mismo ya que el gesto “arpegio” es un acorde cuyas notas son tocadas desplegadas en un tiempo muy cercano. Para hacer esto, la solución tiene varios pasos a seguir. Además del análisis previamente explicado, el programa debe comparar la disposición de las notas del acorde entrante con el acorde ensamblado del banco de notas previamente generado. Este operación se lleva a cabo restando un acorde con otro para así encontrar la menor diferencia en distancia de intervalo de un acorde con otro, entonces el acorde generado es permutado con el objeto “pcs_perm” hasta que encuentra la permutación con menor diferencia. Esto quiere decir que las alturas del acorde que reemplaza el ingresante, es elegido de acuerdo a la mayor semejanza que tenga en la disposición de sus notas y el ámbito que abarca. El programa no analiza mas en profundidad si existen varias permutaciones con igual resultado, elige el primero que tenga la menor diferencia para optimizar tiempo de proceso Por último, el programa busca la mejor acomodación de octava con un algoritmo que establece que si la diferencia en semitonos entre dos notas es mayor a 6, entonces el intervalo puede ser invertido para acercar mas un acorde a otro. [Tabla 1. Ejemplo del algoritmo, se restan los 2 PCS y se suman los resultados]

3		5		7		10		Desde la mesa	3		5		7		10		Desde la mesa
0		4		7		11		Desde el “pool”	4		0		11		7		Desde el “pool”
3	+	1	+	0	+	1	=	5 (mejor opción)	1	+	5	+	4	+	3	=	13

Figura 29. Tabla 1: Dos permutaciones y la elección más apropiada.

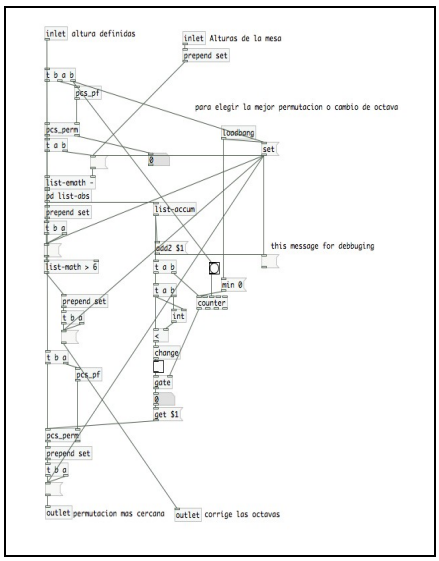

Figura 30. Subpatch que busca la mejor permutación.

5.2.2. Mapeo del gesto “glissando” y “tremolo” a los PCS

Los gestos “glissando” y “tremolo” fueron resueltos de igual manera ya que reciben igual los datos desde el archivo XML. Este, entrega una altura inicial y una altura final, las cuales deben ser acomodadas para que reproduzcan correctamente las alturas de acuerdo al campo armónico elegido. Para ello, el algoritmo de permutación explicado en el proceso anterior es el mismo, pero fue simplificado y realizado sin el objeto “pcs_perm” ya que este necesita tres notas como mínimo para operar y los dos gestos entregan solamente dos. Para lograrlo, se busca el intervalo entre las dos notas extraídas del XML y se busca el intervalo entre las dos notas extraídas del campo armónico. A continuación se busca la diferencia entre los dos resultados y si este supera una cuarta aumentada, se invierte para buscar su inversión y así acercarse mas al gesto inicial ingresado en la superficie.

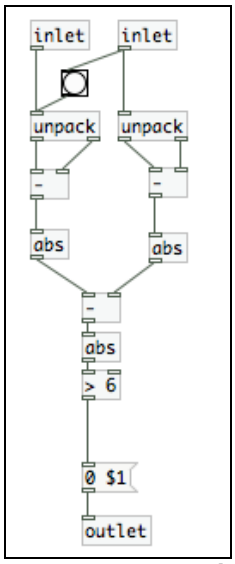

Figura 31. Programa que permuta el "tremolo" y el "glissando".

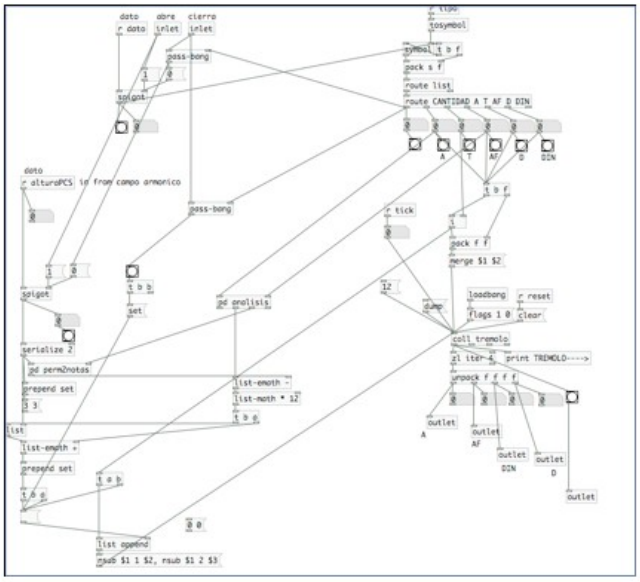

Figura 32. Subpatch con el corrector de alturas implemenetado. Gesto "tremolo"

5.2.3. Mapeo del gesto "melodía" a los PCS

El mapeo del gesto "melodía" a los PCS previamente designados en el campo armónico fue realizado en varias etapas. Para ello, se debe entender primero el concepto de "contorno melódico" y como el algoritmo mantiene su diseño y direccionalidad sin importar si el ámbito en el cual se desarrolla es distorsionado. Esto es necesario, ya que para que el dibujo del contorno se mantenga, es mas importante mantener las direcciones de los intervalos antes que la cercanía de las notas.

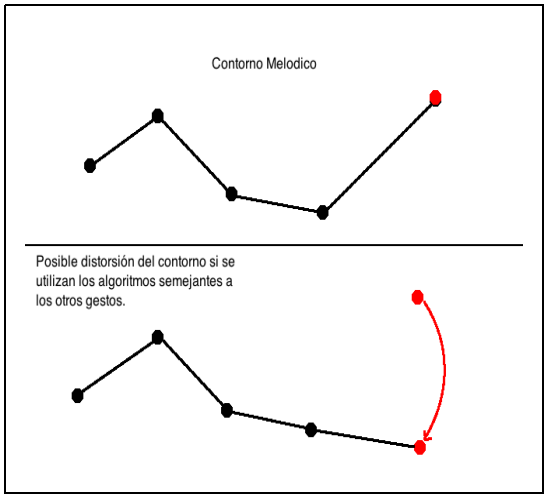

Figura 33. Diagrama de la posible distorsión en el gesto "acorde"

Para lograr esto, se establece una matriz donde los dos PCS distintos, uno entrante de la superficie y otro del campo armónico, se ordenan de acuerdo a un índice designado por la posición en la melodía y se reacomodan de acuerdo a un ordenamiento de menor a mayor. Primero, se asigna un índice de acuerdo al orden proveniente de la melodía ingresada por el XML. Por ejemplo, si entra el PCS [3 0 7 9 5], se crea una matriz asignando un índice de orden a cada una de las notas, produciendo una matriz de la siguientes características.

0	1	2	3	4	Índice (ordenado)
3	0	7	9	5	PCS de la superficie

Posteriormente se lo reordena de menor a mayor para luego poder compararlo con el PCS ingresante desde el campo armónico.

1	0	4	2	3	Índice (desordenado)
0	3	5	7	9	PCS de la superficie

Ahora podemos ingresar el PCS proveniente del campo armónico, ordenarlo de menor a mayor e ingresarlo a la matriz. Por ejemplo, el PCS proveniente del campo armónico va a ser el [4 11 6 2 8] que al ordenarlo quedara de la siguiente manera: [2 4 6 8 11]. Ahora puede ingresar a la tabla y la matriz quedara de la siguiente manera:

1	0	4	2	3	Índice (desordenado)
0	3	5	7	9	PCS de la superficie
2	4	6	8	11	PCS del Campo Arm.

De esta forma tenemos los dos PCS ordenados de menor a mayor y el índice del orden melódico desordenado. Entonces lo único que falta es reordenar la matriz de acuerdo a la primera fila ordenada de menor a mayor y extraer la fila numero tres. Dando como resultado lo siguiente.

1	0	4	2	3	Índice (desordenado)
0	3	5	7	9	PCS de la superficie
2	4	6	8	11	PCS del Campo Arm.

→

0	1	2	3	4	Índice (ORDENADO)
3	0	7	9	5	PCS de la superficie
4	2	8	11	6	PCS del Campo Arm.

El PCS proveniente del campo armónico ordenado correctamente para que mantenga el contorno melódico es el [4 2 8 11 6]. En cuanto al dibujo de la melodía, el resultado el mantenimiento del contorno y así producir la menor cantidad de distorsión del dibujo ingresado desde la superficie.

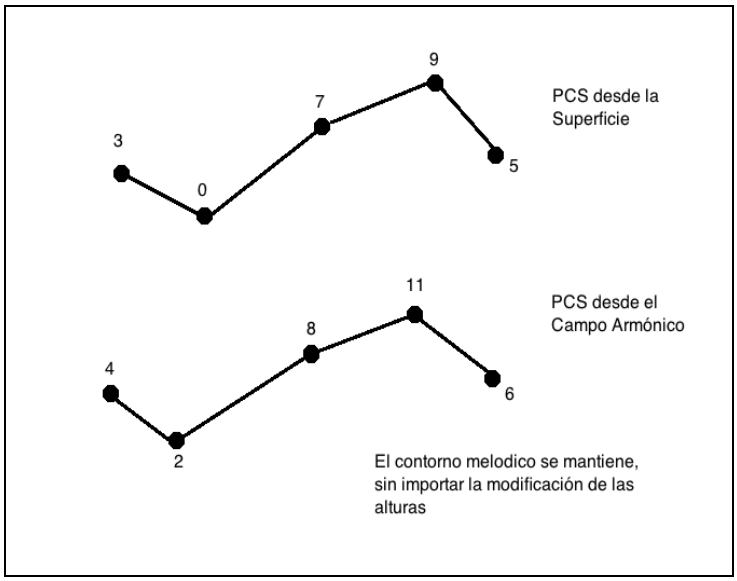

Figura 34. Diagrama de melodia

Toda esta parte del programa se desarrollo en la “abstracción” llamada “zzzzmelodia” [fig] y utilizan el objeto “matrix” incluido en la librería “iemmatrix”. Por ultimo, las nuevas alturas son sustituidas en el objeto “coll” que posteriormente son reproducidas.

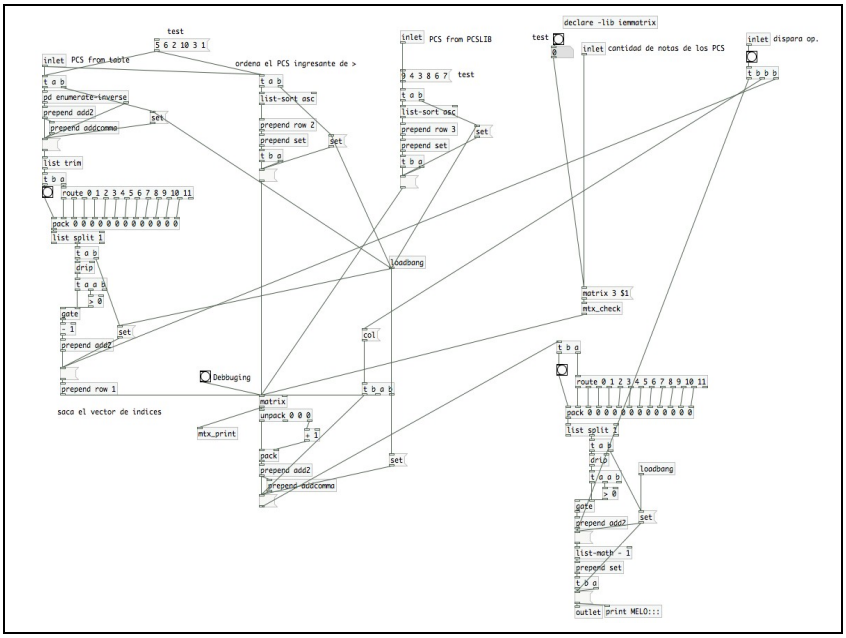

Figura 35. Algoritmo de corrección melódica implementado en PD

5.2.4. Mapeo inexistente del gesto “Nota” a los PCS

El gesto “nota” fue el único que quedo sin un mapeo a una altura designada previamente por el campo armónico. Esto se debe a que la independencia que tiene “nota” como gesto y su relación inmediata a una altura, llevo a la conclusión que era el único gesto totalmente libre del campo armónico.

6. Conclusión

Durante el desarrollo de la pantalla sensible al tacto con aplicación musical conocida como Ugarit, el equipo se planteo objetivos claros de funcionamiento que condicionaron la elaboración de programas especializados. La idea base del proyecto establecía claramente el camino y las dificultades que se iban a encontrar. Ugarit es una mesa táctil que plantea un uso especifico de las tecnologías de realidad aumentada con una visión clara y original de las problemáticas y necesidades actuales para el desarrollo de un aparato de estas características. El concepto marca notoriamente la diferencia entre ésta y otras aplicaciones o dispositivos similares.

En el transcurso de su elaboración, el equipo se encontró con problemas de diseño de hardware y software. El difícil balance que se plantea entre un la tecnología abierta y de bajo costo y la correcta calibración de un sistema de estas características es un reto para cualquier grupo de desarrolladores de este tipo de proyectos. Por eso, el planteo estructural del diseño de hardware y software fue clave para el éxito obtenido. Para la construcción de la mesa se busco tener un equilibrio entre el tamaño de la superficie y una superficie confortable para los usuarios. Era de suma importancia contar con un área de uso lo suficientemente grande para poder interactuar cómodamente, pero que mantenga una porte razonablemente chico. El diseño y armado de la mesa fue documentado de mejor manera posible para que cualquier persona pueda reproducir su construcción en cualquier lugar del mundo.

Así mismo, los obstáculos superados a nivel de software son un logro. La difícil tarea de cualquier aparato que censa movimiento y figuras a través de un sistema óptico, como el utilizado, requiere una precisión extrema. Para lograr esto, el equipo debió afrontar problemas serios como la discriminación de ruido en la imagen censada para eliminar errores severos en la aplicación. Una vez logrado esto, cualquier tipo de mejora al programa es mas fácil de realizar. Además, el beneficio de tener este problema resuelto dio total libertad para resolver problemas mas fáciles como la identificación y clasificación de gestos musicales.

Por ultimo, el proyecto encara una problemática nueva en el tratamiento de estructuras musicales de altura. Una visión original a este punto produjo nuevos desafíos en el desarrollo de la aplicación. La manipulación de vectores y matrices de alturas musicales se logró gracias al estudio y dominio de una librería especializada para ello en el entorno de programación utilizado. La secuenciación de eventos musicales planteo otro reto al equipo, sobre todo por la necesidad de mantener un archivo estructurado y ordenado como el XML utilizado. Una vez desarrolladas las soluciones para estos problemas, la mesa obtuvo un sistema musical robusto.

Ugarit es una mesa sensible al tacto que goza de un sistema solido. El desarrollo de este, aunque se encuentre en una etapa inicial y experimental de prototipo, se logró con solidez y claridad. Esto permite que la aplicación pueda mejorar en futuras versiones gracias a la base estable y robusta conseguida en esta primera etapa de desarrollo.

7. Referencias bibliográficas

[1] Definicion.De, “sesion”, [en linea]. Disponible en la web: http://definicion.de/definicion-de-sesion/

[2] Wikipedia, “Ordenamiento de Burbuja”, [en linea]. Disponible en la web: http://es.wikipedia.org/wiki/Ordenamiento_de_burbuja

[3] Wikipedia, “Ordenamiento de Burbuja”, [en linea]. Disponible en la web: http://es.wikipedia.org/wiki/Ordenamiento_de_burbuja

[4] Wikipedia, “Ordenamiento de Burbuja”, [en linea]. Disponible en la web: http://es.wikipedia.org/wiki/Ordenamiento_de_burbuja

[5] Di Liscia, Pablo, Cetta, Pablo, (2008): Elementos de Contrapunto Atonal, EDUCA, Universidad Católica Argentina, Buenos Aires, Argentina. En prensa.

[6] Di Liscia, Pablo, Cetta, Pablo, (2011): Composición asistida en entorno PD, Revista de Investigación Multimedia (RIM), Instituto Universitario Nacional del Arte, Buenos Aires, Argentina.

[7] Di Liscia, Pablo, Cetta, Pablo, (2011): Composición asistida en entorno PD, Revista de Investigación Multimedia (RIM), Instituto Universitario Nacional del Arte, Buenos Aires, Argentina.

[8] Puckette, Miller(2009),“PD Documentation”, http://crca.ucsd.edu/~msp/Pd_documentation/

[9] Volpe, Gualterio.(2003): Computational models of expressive gesture in multimedia systems, InfoMus Lab.- Universidad de Génova, Italia.

[10] Ernesto Damiani, Jechang Jeong, (2009): Multimedia Techniques for Device and Ambient Intelligence

[11] Michael Haller, (2006): Emerging Technologies of Augmented Reality: Interfaces and Design

[12] Daniel Shiffman, (2008): Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction, Burlington, USA

[13] Lenguage de programación Processing <http://www.processing.org>

[14] Protocolo TUIO, <http://www.tuio.org/?processing>.

[15] Proyecto Reactable, <http://mtg.upf.es/reactable/>.

[16] Reactivision, <http://reactivision.sourceforge.net/>.

[17] Documentación ARToolKit, <http://www.hitl.washington.edu/artoolkit/documentation/>.