PROYECTO FINAL

Sorcerer's Apprentice: un Interfaz para Interacción Gestual Control Multimedial a través de Kinect

Tutor: GÓMEZ, Silvia

Integrantes

Pagliaricci, Francisco

Pomerantz, Alan

Marzo 2015
I.T.B.A - Argentina

ÍNDICE

1. INTRODUCCIÓN

El avance de la tecnología en los últimos años trae consigo innovación y nuevas alternativas a la hora de interactuar con aplicaciones.

Cada vez más se tiende a que el usuario interactúe con los dispositivos a través de su propio espacio, esto es, usando sus manos, sus gestos faciales, etc. Ya se estima que en el futuro cercano, una computadora constará de una central mínima de procesamiento, el resto estará en la nube, con un tamaño más pequeño que de un teléfono celular, y todos sus periféricos se virtualizarán mediante la interpretación de los gestos del usuario, captados por alguna cámara o sensor infrarrojo.

Ahora es posible, mediante la utilización de diversos dispositivos sensores, captar posiciones estáticas y control gestual, y traducirlos en acciones de utilidad, otorgando nuevas posibilidades de aplicaciones que hagan uso de esta novedosa forma de interacción.

El fuerte interés surgido en éste área trajo consigo mejoras y evolución en cuanto a la calidad de sensores, rango de alcance y apertura, velocidad de procesamiento y capacidad, obteniendo cada vez un mejor rendimiento y una mayor precisión de lo captado. La industria del hardware provee una serie de dispositivos capaces de capturar el movimiento.

Dentro de la gama de dispositivos reconocedores podemos encontrar al Kinect.

Kinect para Xbox 360, o simplemente **Kinect** (originalmente conocido por el nombre en clave «Project Natal»),[2] es «un controlador de juego libre y entretenimiento» creado por Alex Kipman, desarrollado por Microsoft para la videoconsola Xbox 360, y desde junio del 2011 para PC a través de Windows 7 y Windows 8.[3]

Kinect, es un dispositivo electrónico accesible, que permite a los usuarios controlar e interactuar con la consola sin necesidad de tener contacto físico con un controlador de videojuegos tradicional, mediante una interfaz natural de usuario que reconoce gestos, comandos de voz,[4] y objetos e imágenes.

Actualmente se ha valorado acerca de las posibilidades de desarrollo montado sobre este accesible equipo, que abarca tanto captura de video como de audio, y que permite desarrollo de aplicaciones multimediales interactivas e intuitivas al usuario.

Como utilidad se presenta en este informe el desarrollo de una aplicación que permite analizar tanto posiciones como gestos, e incluso comandos de voz, en tiempo real, para responder con acciones dentro de una computadora. Es objetivo de este desarrollo

demostrar la capacidad del dispositivo y disparar el mundo de opciones posibles a realizar en cuanto a aplicativos gestuales.

Para poder demostrar el concepto, se ha seleccionado la aplicación *VLC Media Player* para la reproducción de videos, ejecutado por el aplicativo montado sobre Kinect en su versión 1.

La estructura del presente informe se divide en secciones.
En la sección dos se discuten y analizan trabajos relacionados, y el funcionamiento propio del Kinect. En la sección tres se analiza el desarrollo del aplicativo en Kinect, y la implementación en Unity. En la sección cuatro se discute acerca de la arquitectura definida para el proyecto. En la sección cinco se estudia la arquitectura del VLC, el reproductor de medios seleccionado. En la sección seis se detalla la interfaz de usuario de la aplicación. En la sección número siete se describen las pruebas, tests y calibraciones realizadas. Por último, las conclusiones y reflexiones del trabajo, junto las posibles mejoras a futuro se describen en la sección ocho.
La bibliografía se encuentra como anexo, en la sección número 9.

2. Captura de imágenes y movimiento

2.1 Trabajos Relacionados

En 1985 Hutchins et al. [1] en su publicación fundacional analizan que cuando un objeto se representa en una computadora, su manipulación a través del dispositivo debe ser lo más similar posible a la manipulación que los usuarios harían sobre ese objeto en el mundo real. Eso da a los usuarios la sensación de una manipulación directa.
Cuando un usuario interactúa con un dispositivo lo hace a través del lenguaje que propone la interface, no el suyo. Cuanto menor es esa distancia, mayor es la sensación de manipulación directa. Por ejemplo, la "distancia" es menor cuando el usuario introduce una fórmula compleja a través de la selección de objetos y conectando la salida de uno con la entrada de otro (al estilo workflow) que cuando tiene que codificar la misma fórmula en un lenguaje de programación por medio de sentencias anidadas.

Con la existencia de aparatos capaces de capturar posiciones, gestos y movimientos, se intenta aumentar esta sensación de manipulación directa y natural para poder realizar aplicaciones interactivas e interesantes.

El control gestual es una novedad que puede ser encontrado en aparatos electrónicos tales como SmartTV, consolas de videojuegos e incluso dispositivos móviles. Dichos aparatos brindan al usuario la posibilidad de realizar acciones mediante gestos. Los gestos dentro de los dispositivos móviles suelen ser predefinidos, en base a la composición de movimientos que resulten sencillos, tanto en su facilidad de ejecución y reconocimiento, para el usuario final.

Encontramos así, por ejemplo, movimientos para encender o apagar, o incluso modificar el volumen actual dentro de un SmartTV que se encuentra reproduciendo algún tipo de archivo multimedial

fig.1 - *Gesture Control* en SmartTV

Podemos encontrar un interesante brochure de la empresa Samsung, mostrando el funcionamiento de sus controles gestuales dentro de éste link, junto a una especificación particular de cada movimiento.

En base a lo establecido por Hutchins, para mejorar la usabilidad de cualquier aplicación, en términos de utilización, la distancia generada entre usuario y dispositivo debería ser la menor posible. ¿Cómo ésta distancia puede ser atacada mediante el control gestual? La respuesta la encontramos en otorgarle al usuario **la posibilidad de guardar sus propios gestos, basados en la preferencia del mismo.**
De esta forma, el usuario podrá configurar la aplicación para disparar sus acciones en base a movimientos personalizados y customizados.

En el mundo de Kinect, encontramos numerosos trabajos realizados en Kinect for Windows Samples *(http://kinectforwindows.codeplex.com/)* y en el Kinect SDK (Versión 1.8 disponible en el sitio oficial de Microsoft) realizados en diversos lenguajes de programación y cumpliendo operaciones simples de prueba de concepto de las capacidades del Kinect.

Sample	C#	C++	VB	WPF	DirectX	Additional information
Adaptive UI	Yes	No	No	Yes	No	Available in 1.8.0
Audio Basics	Yes	Yes	Yes	Yes	Yes	Available in 1.7.0
Audio Capture Raw	No	Yes	No	No	No	Available in 1.7.0
Audio Explorer	No	Yes	No	No	Yes	Available in 1.7.0
Background Removal Basics	Yes	Yes	Yes	Yes	Yes	Available in 1.8.0
Basic Interactions	Yes	No	No	No	No	Available in 1.6.0
Color Basics	Yes	Yes	Yes	Yes	Yes	Available in 1.7.0
Coordinate Mapping Basics	Yes	Yes	No	Yes	Yes	Available in 1.7.0
Controls Basics	Yes	No	No	Yes	No	Available in 1.7.0, Requires Kinect Interaction
Depth Basics	Yes	Yes	Yes	Yes	Yes	Available in 1.7.0
Depth	No	Yes	No	No	Yes	Available in 1.7.0
Depth with Color	No	Yes	No	No	Yes	Available in 1.7.0
Face Tracking	Yes	No	No	Yes	No	Available in 1.7.0, Requires Face Tracking
Face Tracking Basics	Yes	No	No	Yes	No	Available in 1.7.0, Requires Face Tracking
Face Tracking Visualization	No	Yes	No	No	Yes	Available in 1.7.0, Requires Face Tracking
Infrared Basics	Yes	Yes	No	Yes	Yes	Available in 1.7.0
Kinect Bridge with MATLAB Basics	No	Yes	No	No	Yes	Available in 1.7.0
Kinect Bridge With OpenCV Basics	No	Yes	No	No	Yes	Available in 1.7.0
Kinect Explorer	Yes	Yes	Yes	Yes	No	Available in 1.7.0
Kinect Fusion Basics	Yes	Yes	No	Yes	Yes	Available in 1.7.0, Requires Kinect Fusion
Kinect Fusion Color Basics	Yes	No	No	Yes	No	Available in 1.8.0, Requires Kinect Fusion
Kinect Fusion Explorer	Yes	Yes	No	Yes	Yes	Available in 1.7.0, Requires Kinect Fusion
Kinect Fusion Explorer Multi Static Cameras	Yes	No	No	Yes	No	Available in 1.8.0, Requires Kinect Fusion
Kinect Fusion Head Scanning	Yes	No	No	Yes	No	Available in 1.8.0, Requires Kinect Fusion
Shape Game	Yes	No	No	Yes	No	Available in 1.7.0
Skeletal Viewer	No	Yes	No	No	Yes	Available in 1.6.0
Skeleton Basics	Yes	Yes	Yes	Yes	Yes	Available in 1.7.0
Slideshow Gestures	Yes	No	No	Yes	No	Available in 1.6.0
Speech Basics	Yes	Yes	Yes	Yes	Yes	Available in 1.7.0
Tic Tac Toe	Yes	No	No	Yes	No	Available in 1.7.0
Webserver Basics	Yes	No	No	Yes	No	Available in 1.8.0
WPF D3D Interop	Yes	Yes	No	Yes	Yes	Available in 1.7.0

fig. 2 - Extracto del listado de Samples del KinectForWindows Sample

Estas pequeñas muestras (samples) se suelen utilizar como punto de partida inicial para el desarrollo de aplicaciones. Los mismos son fácilmente integrables, y toma segundos nomás poder estar ya interactuando y hacer uso de las funcionalidades de Kinect.

Como requerimientos básicos, se necesita tener instalado Visual Studio (2010 o 2012), Kinect for Windows SDK y Kinect for Windows Toolkit.

Buscando también en centros de repositorios como GitHub y BitBucket hemos encontrado un proyecto de la compañia LightBuz (*https://github.com/LightBuzz/Vitruvius/*) denominado Vitruvius. Se utiliza este proyecto como base para la confección del presente trabajo, ya que provee simplificaciones de algunas operaciones básicas de reconocimiento, de manera sencilla y directa.

2.2 Estructura del Kinect

La cámara Kinect se puede dividir en dos partes principales una base y un cuerpo alargado unidos por un motor. En el cuerpo de la Kinect encontramos una cámara RGB (Red Green Blue), un conjunto de micrófonos y un sensor de profundidad, en la siguiente figura se muestran los distintos elementos. Gracias al sensor de profundidad la información recibida es en tres dimensiones. Las dimensiones de la cámara son 28,3 cm. de largo 7,5 cm. de ancho y 7 cm. de alto.

fig.3 - Estructura del Kinect

La cámara RGB, detecta los tres colores primarios y tras un código (Rojo, Verde, Azul), que indica el porcentaje de cada color, se determina así el color del píxel, la cámara es capaz de capturar 30 imágenes por segundo. La cámara tiene 640x480 píxel en 32-bit

fig. 4 - Estructura del Kinect (cont)

El sensor de profundidad se compone por un puerto infrarrojos, y un sensor CMOS (Semiconductor de Óxido de Metal Complementario) monocromo. El sensor tiene un rango de entre 1,2 y 3,5 metros de profundidad y un rango angular de 57 grados en horizontal y 47 grados en vertical. El motor acoplado a la base permite un rango de movimiento entre 27 grados positivos y negativos. La tecnología CMOS se basa en una red de semiconductores distribuidos en forma de matriz, en cada celda de la matriz, en función de la cantidad de luz, se acumulará una carga eléctrica. La cámara de proximidad tiene 16 bit y una matriz de 320x240 píxel. El emisor de infrarrojos emite luz la cual tras rebotar en los objetos será captada por el sensor con tecnología CMOS. El conjunto del sensor de proximidad proporciona un cuarto valor al dato de cada píxel, caracterizando de este modo un dato de la Kinect en RGBX

La Kinect es una *depth camera*. Las cámaras normales captan la luz que se refleja en los objetos en frente de ellas. Convierten esta luz en una imagen que se parece a lo que nosotros vemos con nuestros propios ojos. La Kinect, por otro lado, registra la distancia de los objetos que están ubicados en frente de ella. Usa una luz infrarroja para crear una imagen (una *imagen de profundidad* o *depth image en inglés*) que no captura como se ve el objeto, sino donde está en el espacio.

El sistema de coordenadas, esencial para la captura y representación de las mismas, se consolida tal cual ilustra la siguiente figura

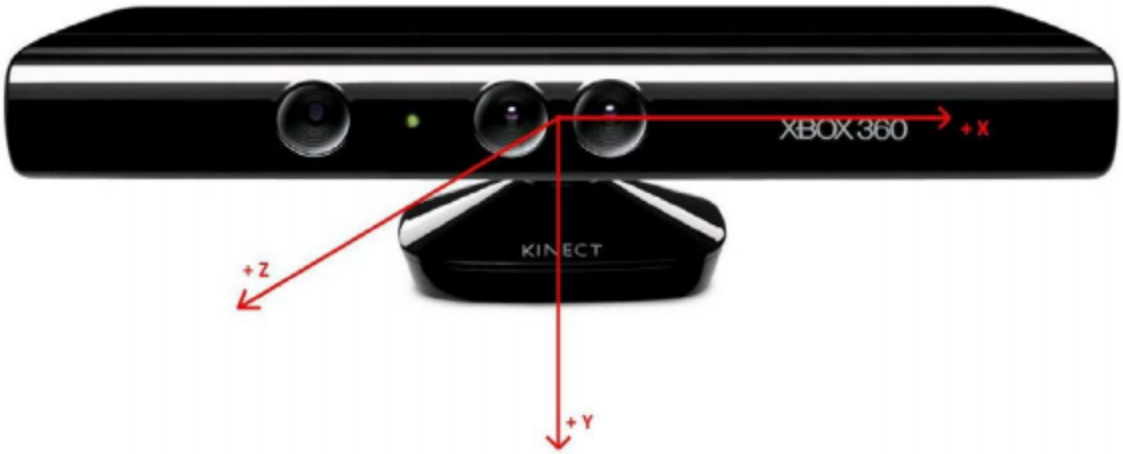

fig. 5 - Sistema de coordenadas en Kinect

2.3 Captura de imágenes

El SDK de Kinect otorga una captura de la posición de lo captado por el sensor de forma que divide al esqueleto en sets de articulaciones.

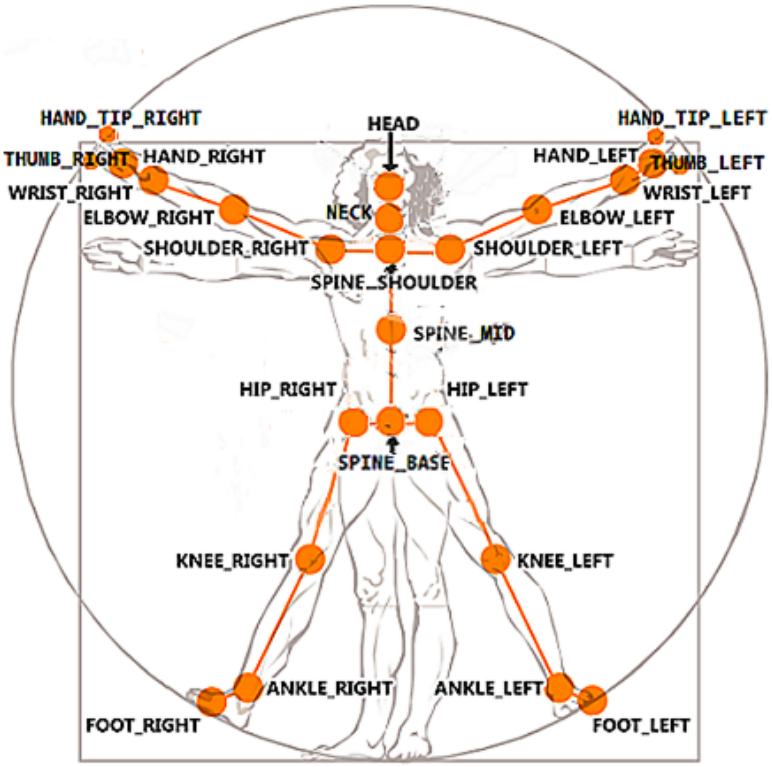

fig. 6 - Esquema de articulaciones para Kinect

El SDK nos proporciona tanto con las posiciones (X,Y,Z) de cada articulación que se esté trackeando, como los cuaterniones de rotación jerárquica o absoluta de los mismos.

2.4 Captura de movimiento

Existen diversas formas de comparar movimientos, para determinar similitud y poder cuantificar el grado de diferencia entre dos series de fotogramas.
Como marco teórico, introduciremos la explicación de dos técnicas, entre otras, que fueron evaluadas, para la comparación de series de fotogramas, para extraer gestos y movimiento.

2.4.1 DTW

El Alineamiento Temporal Dinámico (Dynamic Time Warping, DTW), es una técnica surgida de la problemática inherente a diferentes realizaciones de una misma serie de tramas, en las que se observa una variabilidad interna en la duración de los grupos visuales que la forman, de modo que no existe una sincronización temporal (alineamiento temporal). Además, esta falta de alineamiento no obedece a una ley fija (p. e., un retardo constante), sino que se da de forma heterogénea, produciéndose así variaciones localizadas que aumentan o disminuyen la duración del tramo de análisis. La problemática asociada hace referencia a la dificultad añadida en el proceso de medida de distancia entre patrones, puesto que se están comparando tramos que pueden corresponder a unidades visuales distintas.

Para trabajar con estas series de tramas, se aplica el algoritmo DTW como sigue:

Consideramos dos secuencias A y B, compuestas respectivamente por n y m vectores.

Cada uno de estos vectores es d-dimensional y puede ser representado como un punto en un espacio d-dimensional. Es importante destacar que la longitud de las secuencias A y B pueden ser distintas.

$$\mathcal{A} = a_1, a_2, \ldots, a_i, \ldots, a_n$$
$$\mathcal{B} = b_1, b_2, \ldots, b_j, \ldots, b_m$$

fig. 7 - Series d-dimensionales

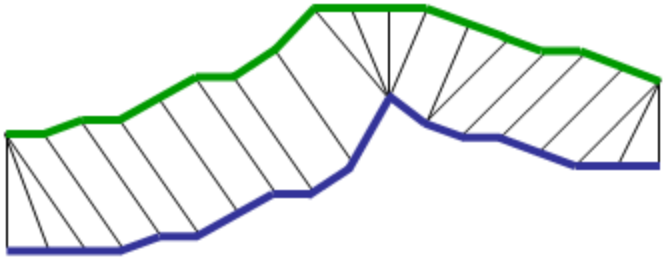

fig. 8 - Alineación de dos series mediante DTW

Dtw funciona bajo la comparación iterativa de dos series temporales, buscando la mejor aproximación entre ambas. Observando la figura superior (Un ejemplo de dos series 1-dimensionales) el eje temporal es distorsionado de manera tal que cada punto dentro de la serie verde es óptimamente alineado a la serie azul.

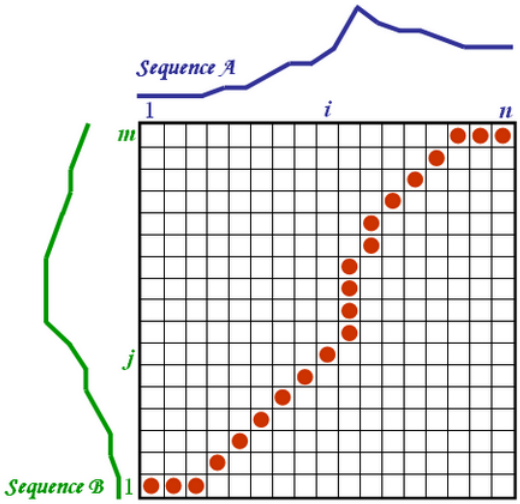

fig. 9 - Matriz de distancias generadas por ambas series

A partir de esto, podemos construir una matriz de distancias de dimensión n x m. Dentro de esta matriz, cada celda (i,j) representa la distancia entre el i-ésimo elemento de la secuencia A y el j-ésimo elemento de la secuencia B. La métrica utilizada comúnmente para mesurar la distancia es la ‘distancia euclídea’. La resolución del algoritmo consta en encontrar el camino óptimo, que puede ser hallado utilizando fuerza bruta, o programación dinámica

2.4.2 Angle Comparison

Para comparar movimientos se puede recurrir a comparar los ángulos formados entre las articulaciones, fotograma a fotograma, y acumular el error de cada aproximación. Basta con tomar la componente X e Y de la imagen, y mediante la función matemática arcotangente

obtener el ángulo formado por dichas componentes. De éste método sencillo y básico, se pueden realizar comparaciones a primer instancia de movimientos, y extraer una conclusión de si hay similitud entre distintas series de fotogramas, definiendo un umbral máximo de aceptación ante sucesivas series de fotogramas.

2.4.3 Método utilizado

DTW fue concebida como una opción, bajo sugerencia de la cátedra, no obstante fue descartada debido al gran trabajo de procesamiento que requiere, el cual se traduce en latencia percibida y un nivel de precisión que no es necesario.

Angle comparison fue la elegida para realizar las comparativas, ya que, a pesar de la simplicidad del método, el mismo arrojó buenos resultados, a nivel bajo de complejidad de desarrollo e integración.

3. Desarrollo de Aplicativo en Kinect

3.0 Enunciado

".. El objetivo de este proyecto es el desarrollo de una interfaz gestual para accionar eventos en una computadora a través de los gestos del usuario captados por una cámara simple o un dispositivo kinect.

Una vez terminada dicha interfaz, debe poder ser utilizada para manejar los controles de audio y video con el movimiento de las manos y los gestos faciales del usuario."

Extraído del enunciado del listado de proyectos

3.1 Definición de Requerimientos

Se definió los requerimientos del aplicativo en Kinect, como un aplicativo tal que otorgue un feedback al usuario del reconocimiento y tracking en tiempo real de las posiciones de la figura de lo captado por el sensor, reconocimiento de gestos preconfigurados y determinados, y la posibilidad de grabar gestos propios. También a su vez, se le incorpora

control por medio de la voz, y la integración con un reproductor de medios (VLC Media Player en su versión 2.2.0) para controlar sus funciones básicas (Volume, Play/Pause, Reproduction Speed).

3.2 Unity

Unity es un motor de videojuego multiplataforma creado por Unity Technologies, que provee al desarrollador de una plataforma de desarrollo flexible y poderosa para crear experiencias interactivos 2D y 3D multiplataforma. Es una de las herramientas más utilizadas en la industria de los videojuegos por su capacidad de integrar comportamiento programable junto con la interfaz visual. El motor de Unity integra una suite de herramientas para las necesidades del desarrollador de videojuegos, incluyendo un motor de física que modela el universo, utilidades de control de sonido y herramientas sofisticadas para modelado y animación 2D y 3D.

Se decide inicialmente la elección de Unity como motor gráfico para desplegar la visualización de lo captado por el sensor Kinect, justamente para aprovechar las herramientas visuales que ofrece Unity, y poder desarrollar una interfaz visual avanzada en un tiempo razonable.
Para modelar la aplicación por medio de Unity, utilizamos las herramientas de UI básicas de la interfaz visual nueva (GUI version 4.6) para implementar los botones y menúes.

Para representar al usuario que está utilizando la aplicación, se utilizó un modelo 3D gratuito descargable del Unity Asset Store (Robot Kyle).

Robot Kyle es un modelo 3D de robot que puede ser utilizado dentro de Unity para realizar animaciones y movimientos. Por medio de la manipulación tanto de la posición, como de la rotación y la escala del modelo, se puede ubicar el robot en la posición deseada. Además, tiene una estructura jerárquica de articulaciones que se utiliza para modificar las distintas partes del robot, simulando así huesos y diversas posiciones del robot.
Sobre esta estructura del robot se ubica un mesh con una textura específica que es lo que permite la visualización del modelo.
Este mesh presenta restricciones por lo que no se pueden definir posiciones arbitrarias para las distintas articulaciones del robot, principalmente porque esto lo deformaria e impediría que sea correctamente dibujado.

3.2.1 Integración Unity-Kinect

Inicialmente se intenta utilizar un plugin para utilizar el Kinect desde Unity (http://channel9.msdn.com/coding4fun/kinect/Unity-and-the-Kinect-SDK) sin embargo, este plugin esta diseñado para la version demo del Kinect SDK previo a la version 1.0, y tiene

problemas de compatibilidad con las versiones actuales (SDK 1.8 & Unity 4.5). Otras integraciones más modernas utilizan el nuevo dispositivo Kinect V2.0 junto con su SDK que provee mayor funcionalidad junto con un gran incremento en la capacidad y calidad del reconocimiento. Estos proyectos exportan distintas funciones *wrapper* que delegan en llamados a distintas funciones del sensor, y con esto construyen una *dll*, que luego importan como plugin en Unity para poder ser utilizada. De esta forma, desde el código en Unity se realiza un llamado a la *dll* propia, que efectivamente termina generando un llamado al SDK de Kinect. Intentamos realizar una implementación propietaria utilizando la misma técnica pero sin embargo nos encontramos con problemas de compatibilidad de Unity, que exige que los llamados a funciones externas sigan estrictas reglas dentro de su configuración interna.

Ante la imposibilidad de utilizar un plugin, surgen otras formas de comunicar los procesos para permitir comunicar la interfaz de Unity. Una primera aproximación a modo *prueba de concepto* fue una implementación que, por medio de lectura y escritura en archivos, transportaba la información (los skeletons serializados, previamente detectados por el sensor). De esta forma, se implementó un server conectado al sensor Kinect, que mantenía actualizado el archivo con la última información brindada por el lector y la aplicación de Unity, que en el update consultaba el archivo para actualizar la posición del robot. Esta implementación no pudo ser utilizada ya que el sistema utilizaba de manera intensiva dicho archivo, y presentaba complicaciones al ser abiertos para lectura y escritura por procesos distintos, de manera concurrente. Descartada este esquema de productor-consumidor sobre un archivo, se decide implementar el server como una *Web API* que contenga la información de la última actualización disponible para consultar desde Unity. Sin embargo, esta implementación fue rápidamente descartada, ya que el overhead generado por las peticiones HTTP en una *Web API* dentro del método update (El método que actualiza los frames del modelo 3D) de Unity generaba una excepción por desincronización.

La decisión finalmente utilizada fue la comunicación a través de Sockets, en el que se implementa la transmisión de los skeletons por parte del server C#.
Al recibir un update desde el SDK de Kinect, el server utiliza la información de los skeletons y crea una representación particular para transmitir por medio del socket. Cualquier cliente que esté escuchando en ese socket va a recibir periódicamente updates de los skeletons. Cabe destacar que el cliente podría ser una aplicación codificada en cualquier lenguaje, particularmente implementamos un cliente en Unity para utilizarlo en la creación de una UI para nuestro proyecto.

3.3 Serialización

Con el objeto de transmitir los Skeletons captados por el sensor de la Kinect en los pasos anteriores, es necesario serializarlos bajo un formato propietario.
La siguiente figura presenta un macro esquema de como es la serialización utilizada.

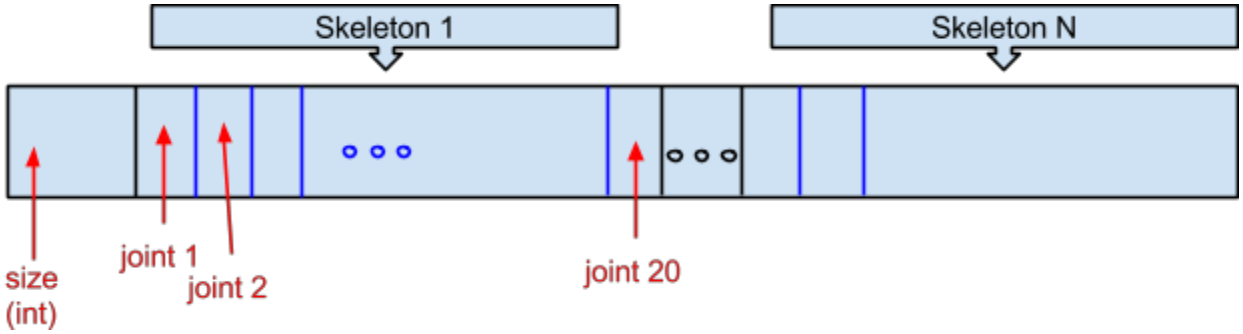

fig. 11 - Serialización de un Skeleton

El SDK de kinect nos presenta una actualización que proporciona un array de objetos de clase Skeleton. Estas instancias representan a las distintas personas detectadas por el sensor en ese momento. Cada skeleton representa la ubicación de una persona por medio de un conjunto de articulaciones o *Joints*, donde para cada articulación se proporciona la posición (x,y,z) en metros y el cuaternión de rotación relativa al padre (más información). El formato de serialización para cada skeleton consiste en transformar cada conjunto de articulaciones como cadenas (strings) siguiendo este esquema:

$$jointType\#pos_x|pos_y|pos_z|rot_x|rot_y|rot_z|rot_w$$

Al haber una cantidad variable de skeletons, el tamaño de paquete no es fijo, por lo que se prefija con un entero que contiene el tamaño en bytes del paquete.

El cliente de Unity va a ejecutar en el update y va a verificar si hay información en el socket. Si el framerate de Unity resulta ser más rápido que el del server, entonces van a haber updates en los cuales el cliente no va a encontrar contenido, cuyo caso se definió ignorar el update (ya que el framerate de Kinect se considera suficiente para una correcta experiencia final y, en los casos en que el socket se encuentra vacío, se concluye que no hay nueva información disponible). En el caso contrario, el framerate de Unity es mas lento que el de Kinect, y en el método update el socket contiene gran cantidad de información, y al ser un sistema FIFO el primer paquete dentro del socket corresponde a la información más antigua.

El problema hallado, es que en primer lugar se experimentó latencia, ya que la información que se consulta esta desactualizada, y el mayor problema es que, teniendo un productor que produce más de lo que es consumido, eventualmente el socket se llenará, bloqueando el procesamiento.
Para evitar este problema, si al leer un update se detecta que en el socket hay más información, se ignora y se consulta el siguiente paquete, y de esta forma se considera solamente la información más actualizada.

3.4 Visualización

Una vez recolectada la información de los skeletons del ultimo update recibido por el socket, se elige el primero como default. Para representar este skeleton se utilizan las rotaciones relativas de cada articulación. El primer problema que se presentó fue que las representaciones 3D de Unity siempre tienen la misma jerarquía de articulaciones, que tiene más elementos que los presentes en la representación de Kinect. Esto conlleva un problema, ya que hay articulaciones para las cuales no se sabe la rotación. Para resolverlo, decidimos deshabilitar las articulaciones que no estén presentes en el modelo de Kinect. Por otro lado, el Kinect tiene un sistema de referencia determinado, y las rotaciones están representadas dentro de ese sistema. Al tener un sistema de referencia distinto en Unity, es necesario realizar una transformación, para evitar que cuando la persona se mueve en un eje, el dibujo se mueva en otro eje. Además, la representación 3D del robot tiene una serie de restricciones, por lo cual hay rotaciones que intentar imponer y son ignoradas porque contradicen las restricciones. Con esto en cuenta, intentamos representar con el robot la información recibida por el Kinect sin éxito, ya que el sistema de referencia nos generó problema y las rotaciones detectadas por Kinect resultaron ser muy inestables, al ser difícil para el sensor determinar las rotaciones de cada articulación.

La practicidad de Unity para modelar lo sensado por Kinect, motivación que nos impulsó a optarlo como opción, no pudo ser hallada, y el modelo visual en Unity tuvo que ser abandonado.

3.4.1 De Unity a WPF

Como se explica en la sección 3.4, y en base a los problemas de conversión y representación dentro del motor de Unity nos empujan a encontrar alternativas para graficar el esqueleto sensado por el Kinect.

Se decide finalmente entonces, utilizar Windows Presentation Foundation (WPF) como front-end para la visualización e interfaz del aplicativo.

Para ello, el primer objetivo fue lograr representar el movimiento de lo sensado por el Kinect, y desplegarlo al usuario final. Se implementa la interfaz como un skeleton unido por puntos que acompaña y representa en tiempo real los movimientos del usuario frente al sensor

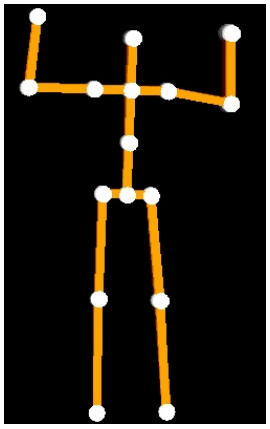

fig. 10 - Representación de un Skeleton en WPF

Las clases dentro del proyecto que se encargan de la representación y visualización del esqueleto son DrawingUtils.cs y SkeletonUtils.cs, que cuentan con los métodos respectivos para lograr dibujar y escalar lineas dentro del *canvas*, dadas ciertas posiciones en el espacio.

La representación final lograda replica los movimientos del usuario frente al sensor, en tiempo real, y con una gran precisión. Sin embargo, el skeleton a veces realiza pequeños saltos, producto de interferencias captadas por el Kinect. De manera empírica observamos que el Kinect bajo ciertas configuraciones de luz no funciona de manera óptima y genera éste tipo de sobresaltos por lo que es necesario generar una fuente de luz uniforme, en el área de proyección, a fin de evitar este tipo de interferencias.

4. Arquitectura

La arquitectura está basada en una arquitectura Cliente-Servidor distribuida (WpfInterface-KinectServer). La comunicación se realiza mediante Sockets divididos por funcionalidad. Encontramos así entonces un canal de comunicación (TcpClient) para la transmisión del vídeo obtenido por el sensor, la transmisión de los comandos de voz, y la transmisión de los *Skeleton* sensados por el Kinect.
La arquitectura elegida, permite separar el sensado y procesamiento, presentes en el servidor, de la aplicación interfaz con el usuario.

4.1 Servidor (KinectServer.sln)

El servidor es el encargado de crear los canales para comunicar cada tipo de dato. Creamos así un skeletonServer, un cameraServer y un voiceServer, que se encontrarán disponibles en los puertos TCP 8081, 8082, y 8083 respectivamente. Cada servidor va a transmitir un tipo de dato distinguido, a medida que reciba información del Kinect.

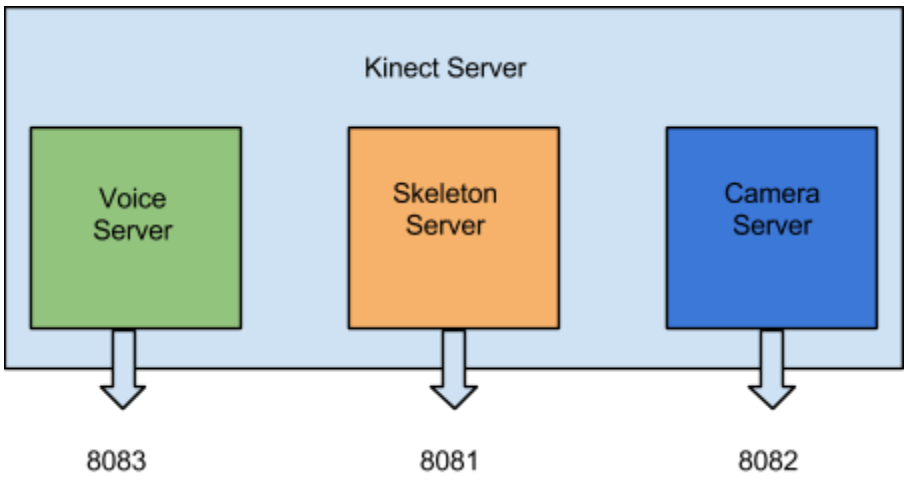

fig. 12 - Esquema de Servidores y sus respectivos puertos

4.1.1 Servidores internos

- Voice server: Reconoce los comandos de voz de la lista de palabras reconocidas (diccionario) e informa cuando una palabra ha sido *detectada*, *hipotetizada* y *reconocida* junto con su respectivo nivel de confianza, dentro del algoritmo de detección.
- Skeleton server: Detecta las posiciones de las personas en el rango de visión de la cámara. Informa a 30fps con un vector de objetos de la clase *Skeleton* con la información de cada esqueleto. Este vector puede estar vacío si no se han detectado personas (figuras humanas) en ese periodo, o puede tener *n* personas, cada una identificada con un id unívoco.
- Camera server: Provee una imagen representando la visión de la cámara en ese momento. Informa a 30fps y provee un vector de bytes representando la imagen captada en formato mapa de bits.

La estructura básica de cada server se maneja de la misma forma que el Kinect, por medio de *callbacks*. Cada vez que Kinect presente una actualización del estado de uno de los sensores, va a informarle al server por medio de un callback, y este va a transmitirle a los clientes la misma información.

Cada server internamente se comporta en forma similar. Tiene un thread dedicado exclusivamente a obtener nuevos clientes, que una vez registrados se van a agregar a la lista

de clientes. En cada callback que recibe nueva información, se les va a transmitir a todos los clientes la actualización con la representación correspondiente a ese tipo. Si en algún momento se detecta un problema con un canal en particular, el server automaticamente lo desuscribe e intenta cerrar, y el cliente debe reconectarse. Si bien esta decisión de diseño tiene baja tolerancia a errores, y en conexiones problemáticas puede generar muchas desconexiones, es un costo pequeño asumido en favor de una casi instantánea actualización a los clientes disponibles para evitar latencia. Esto permite correr el cliente visual en otra terminal dentro de la misma red con tan baja latencia que pareciera estar conectado directamente al sensor

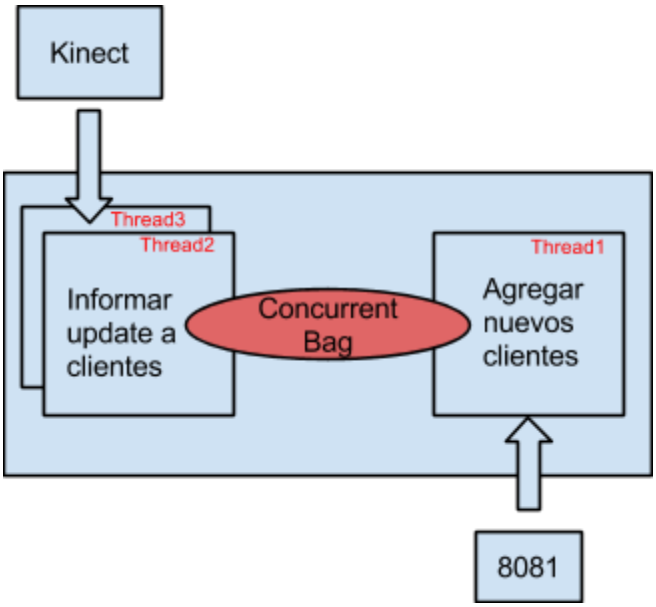

fig. 13 - Esquema de manejo de clientes

La distribución del servidor principal en múltiples servidores internos dedicados exclusivamente a un tipo de dato permite una implementación simplista de los clientes. Al escuchar todos los servers en un puerto distinto, uno puede implementar un cliente al que solamente le interesa recibir la información inherente a los *skeletons,* por ejemplo, registrando en el puerto 8081 y teniendo la seguridad de que se evita el overhead de procesar información que no es de interés, y por otro lado asegurando que, cuando un nuevo paquete arriba, la representación es estándar y el tipo de dato es el esperado. En el caso de interesarse sólo en skeletons, basta con conocer su representación serializada para registrarse y comenzar a recibir paquetes deserializando sin miedo a excepciones, sin tener ningún conocimiento de los otros tipos de información disponibles, ni de sus representaciones.

4.2 Cliente (WpfInterface.sln)

Dentro del cliente podremos encontrar todo lo referido a la manipulación de la información sensada por el Kinect, los comandos de voz deducidos, y el manejo del reproductor VLC.

En primera instancia, el cliente cuenta con una interfaz gráfica que provee al usuario de un menú de opciones de configuración , y las diversas acciones posibles a realizar. La interfaz es detalladamente explicada en la sección **6. Interfaz de Usuario.**

Como ya se indicó, del lado Servidor se han creado respectivos TCPClient para cada funcionalidad (skeletonServer, cameraServer, voiceServer). Conforme a esto, del lado cliente, tienen que encontrarse los respectivos skeletonClient, voiceClient y cameraClient, los cuales se conectaran a la combinación ip:puerto provista por el parámetro de configuración (Set Server IP) o en caso contrario, se utilizará el valor default provisto por el programa como localhost (127.0.0.1) asumiendo que la misma máquina es Servidor y Cliente a la vez.

Cada TCPClient maneja, como ya se ha dicho anteriormente, de manera interna una ConcurrentBag (System.Collections.Concurrent) en donde manipula, mediante los métodos ‘subscribe’ y ‘unsubscribe’ los distintos *listeners* que recibirán la información enviada por el servidor.

Modelar de esta forma además permite que los *listeners* puedan ser removidos o agregados en tiempo de ejecución, permitiendo *hot-swapping* de los mismos. Esto desemboca en la posibilidad de modificar el servidor o las terminales VLC que se están utilizando en *runtime*

4.3 Consideraciones de implementación

4.3.1 Arquitectura

Al tratarse de una arquitectura de servidor, fue necesario tener algunas consideraciones para lograr una implementación eficiente. En primer lugar, para permitir suscribir clientes en cualquier instante, se creo un thread aparte que cicla constantemente a la espera de nuevas conexiones. Una vez llegada una nueva conexión, se realiza el handshake y se establece un canal para comunicarse, y este canal se agrega a la lista de clientes y el thread vuelve a dormirse a la espera de otro nuevo cliente. En segundo lugar, se debe considerar que la lista de clientes está siendo utilizada constantemente, principalmente en iteraciones de actualización, por lo que se requirió utilizar una colección concurrente.
Para lograr ello, utilizamos un ConcurrentBag, para evitar una excepción, o que se salteen elementos en la iteración. En tercer lugar, se debe tener en cuenta el costo de las operaciones. No se deben realizar operaciones muy costosas o bloqueantes porque impedirían un trato equitativo de clientes, teniendo clientes que reciben una actualización primero y otros que la reciben sustancialmente después. Inclusive, al ser manejado por callbacks, es posible que mientras se esté informando de un update llegue otro nuevo, y se

empiece a enviar en paralelo al anterior. Finalmente hay que tener en cuenta que pueden ocurrir fallas durante la transmisión, y que el servidor debería ser tolerante a fallas en canales particulares e impedir que una mala conexión lo deje sin funcionamiento o afecte a otros clientes. Esto se refleja en la mencionada desuscripción en caso de existir cualquier excepción en la comunicación.

4.3.2 Interfaz del server

El server posee una simple interfaz visual creada en WPF. Si bien la funcionalidad central se basa en transmitir la información, decidimos agregarle una simple interfaz visual, para que le permita rápidamente al usuario ver lo que esta detectando el sensor y permitirle reubicarse o cambiar la posición del Kinect. Además, para facilitar la ubicación, permitimos interactuar con el motor que define el ángulo de elevación de la cámara. De esta manera, si ve que lo que está captando la cámara no es lo deseado puede cambiar el angulo de elevacion interactuando con la slide bar de la interfaz.

Una correcta configuración de la elevación del sensor, es aquella que permite visualizar una total extensión de los brazos, en posición vertical, y dentro del foco de la cámara.

fig. 14 - Incorrecta posición del ángulo de elevación de la cámara (Los brazos no entran en el foco de la cámara)

fig. 15 - Correcta posición del ángulo de elevación de la cámara (+9º para el ejemplo)

El motor de elevación viene con grandes limitaciones de movimiento, que permiten una cantidad limitada de cambios por segundo, por lo que la barra se bloquea por un segundo luego de un ajuste y se vuelva a habilitar al cabo de ese tiempo. Si bien esta interacción puede resultar un poco tediosa, solo se utiliza en la etapa de configuración y está marcada por las limitaciones del sensor en la versión 1 del Kinect.

También el servidor es quien define la lista de comandos de voz (Ver Lista de Comandos en **6.1 Lista de Comandos de Voz**) a ser reconocidos por la aplicación y los agrega a su propio diccionario de reconocimiento. Es importante indicar que el reconocimiento de los comandos de voz es una tarea realizada por el propio servidor, y lo que envía al cliente es ya la sentencia del comando seleccionado. Esta consideración de diseño fue determinada de forma arbitraria en la etapa inicial del desarrollo.

4.3.3 Arquitectura de *listeners*

Como fue mencionado anteriormente, esta aplicación recibe los updates del servidor de Kinect y en consecuencia realiza acciones e interacción con el usuario. Para esto, crea una conexión por medio de una instancia de TcpClient para recibir las actualizaciones por medio de los paquetes que llegan al socket. Siguiendo la arquitectura del server, uno podría crear múltiples conexiones con el servidor y recibir actualizaciones en cada módulo de acuerdo a sus necesidades, sin embargo esto no sería eficiente por la cantidad de conexiones generadas.La solución correcta es crear una unica conexion por cada tipo de dato y que cada módulo tenga una referencia a la instancia que está manteniendo la conexión. Sin embargo, y dado la naturaleza con la que funciona el sistema, donde recibe la información del servidor y responde de acorde, se creó una estructura inversa, por medio de una implementación orientada a eventos, en donde a los módulos se les comunica que se recibió información nueva, y cada módulo responde de acorde a la información recibida.

Se define entonces una interfaz llamada ClientListener con un unico metodo que es dataArrived(object data). Cualquier clase que quiera recibir información del server tiene que implementar esta interfaz, y cuando se suscriba al cliente (TcpClient) va a comenzar a recibir los *updates*. El manejo de la información es propio de cada implementación y su propósito particular, y no forma parte de la focalización del TcpClient. Si observamos el método dataArrived, éste recibe una instancia de object, ya que el cliente no conoce el tipo de dato que está recibiendo, solo recibe información y la comunica a los interesados.

Para la suscripción de los ClientListeners tenemos los métodos subscribe(ClientListener listener) y unsubscribe(ClientListener listener). Por las mismas razones descritas en el server, utilizamos una instancia de *ConcurrentBag* para manejar los listeners y permitir iterar por la lista y en otro *thread* borrar o agregar elementos sin corromper la integridad de la misma.

Por otro lado tenemos el manejo de la información, que se realiza por medio de los métodos *tryRun* o *runLoop.* El primero verifica si hay información en el socket e informa a los listeners en caso de tenerla y el segundo corre infinitamente informando y solamente se detiene si se recibe una llamada a *shutdown*. Básicamente cada ciclo verifica si tiene información el socket, la recibe, la deserializa a un object y recorre la lista de listeners suscritos para informarles el dato recibido. Cabe resaltar que, como se dijo previamente, la implementación del TcpClient no conoce del tipo de datos que recibe (y por eso informa por medio de un object) y por eso puede ser utilizada para conectarse a cualquiera de los puertos del server. Por esta razón, en la implementación del cliente hay tres instancias de TcpClient, cada una conectada a un puerto distinto del server y cada una informando a sus respectivos listeners. Cada instancia de listener es responsable de suscribirse al cliente correcto, y de transformar la información de acuerdo al conocimiento que tiene del dato que recibe.

Esta arquitectura es lo que permite incluir nuevos modulos que analizen los datos enviados por el server en forma simple, rápida y tolerante a fallas. A continuación se detallan los distintos listeners utilizados por la aplicación y se puede observar este comportamiento. Algunos se suscriben únicamente en el momento necesario y una vez que completan su objetivo se desuscriben.
Otros como es el caso de la comparación de movimientos crean una instancia del listener para cada movimiento particular y los suscriben y desuscriben según se habiliten/deshabiliten.

- **SkeletonListener:** Se suscribe al skeleton client y en el update, recibe la lista de skeletons, selecciona el default y lo dibuja sobre la UI. Es el encargado de mostrar el esqueleto en la pantalla para que refleje al usuario.
- **CurrentRecording:** Se suscribe al skeleton client en cuanto se inicia el grabado de movimientos y en cada update agrega el default skeleton a su instancia de SkeletonRecorder. Se desuscribe en cuanto se presiona stop, y se utiliza el recording para guardar el movimiento grabado.
- **RecordingReproducer:** Se suscribe al skeleton client en cuanto se requiere reproducir un movimiento grabado. No utiliza la información recibida, sólo utiliza la llamada para marcar el tiempo del update, y así lograr reproducir el movimiento a la velocidad original. Una vez terminado el movimiento, simplemente se desuscribe del client. Se va a crear una instancia por cada movimiento que se quiera reproducir
- **PositionAnalyzer:** Se suscribe al skeleton client y en cada update analiza la posición de un brazo y en función de la posición ejecuta las acciones. Se va a utilizar una instancia por cada brazo.
- **MovementAnalyzer:** Se crea con un movimiento particular y lo compara con los datos recibidos por medio del skeleton client. Para esto, crea un SkeletonRecording de tamaño fijo para utilizar como stream de datos. Este recording que tiene tamaño igual al largo del movimiento va a funcionar como una ventana que en cada momento contiene únicamente los últimos *n* skeletons y los usa para comparar contra el movimiento y definir si fue correcto. Se va a utilizar una instancia por cada movimiento que se quiera trackear y se suscriben al cargarse el movimiento y se desuscriben al deshabilitarse.
- **VoiceListener:** Se suscribe al voice client y en cada update, recibe el comando hablado, lo compara contra el threshold y si el nivel de confianza es aceptado se ejecuta la acción correspondiente. Si nivel de confianza es muy bajo, el comando se ignora.
- **CameraListener:** Se suscribe al camera client y en cada update actualiza la que muestra la visión de la cámara. Al requerir gran cantidad de información transmitida (la imagen completa grabada por la cámara) puede resultar en lag si se envía esa información por la red. Para evitar el lag es necesario un complejo sistema de streaming que esta fuera del alcance de este proyecto, por lo que fue deshabilitado, pero puede ser utilizado si se posiciona el client y server en la misma terminal.

4.3.4 Estructura de acciones

Como se explicó en la sección anterior, se tiene una única clase para los movimientos y son las distintas instancias de esa clase las cuales se suscriben y desuscriben para trackear los movimientos y ejecutar las distintas acciones en caso de ser detectados.
Para lograr esto, se creó la interfaz Action, con un unico metodo void perform(). Esta interfaz es implementada por distintas clases, cada una representando una acción a realizarse, en este caso comandos para todas las instancias del reproductor VLC, *faster*, *normal* y *slower*. De esta forma, al crear una instancia de *MovementAnalyzer* se reciben dos parámetros un *SkeletonRecording* representando el movimiento que se quiere detectar y una instancia de *Action* a ejecutarse cuando ese movimiento se detecte. Esto nos proporciona una gran abstracción para fácilmente incorporar nuevos movimientos y también nuevas acciones ya que simplemente se debe crear una clase que con el método perform() y en ella se puede ejecutar cualquier código sin restricciones.

4.3.5 Persistencia de Gestos

Como se mencionó anteriormente, los movimientos se registran por medio de instancias de SkeletonRecording. Esta clase registra en una lista las posiciones recibidas en cada actualización y provee funcionalidad para su reproducción. También posee funcionalidad para crear movimientos de longitud fija y permitir usar la instancia como un stream de datos con una ventana que muestra los n últimos datos, que es utilizado para comparar movimientos. La persistencia de los movimientos se realiza por medio de la serialización de esta lista, que está formada por instancias de Skeleton del SDK de Kinect que afortunadamente son serializables. Con estos datos serializados, se crea un simple archivo binario que contiene la información. Para la carga desde el archivo, solamente es necesario deserializar el binario a una lista de skeletons y la representación ya esta completa.

4.3.6 Comparación de posiciones

Para la detección de buckets se utiliza las posiciones de los brazos del usuario. El SDK nos proporciona tanto con las posiciones (X,Y,Z) de cada articulación como los cuaterniones de rotación jerárquica o absoluta. Como ya fue mencionado en la sección 3.3, nos encontramos con problemas con el uso de cuaterniones debido a la gran cantidad de ruido que se nos presentó, y por esta razón decidimos utilizar las posiciones. El problema de utilizar las posiciones absolutas es que solo funcionan para una persona específica en una configuración específica (no funcionaria la misma configuración si uno se acerca o se aleja del sensor). Para evitar estos problemas decidimos calcular por nuestra cuenta ángulos entre las articulaciones, pero valiendo de ciertas simplificaciones para evitar la aparición de ruido.

Para detectar las distintas posiciones, se toma en cuenta el ángulo absoluto que se forma entre el codo y la muñeca de cada brazo, que son los que van a determinar en forma más precisa el ángulo en el que se está apuntando.

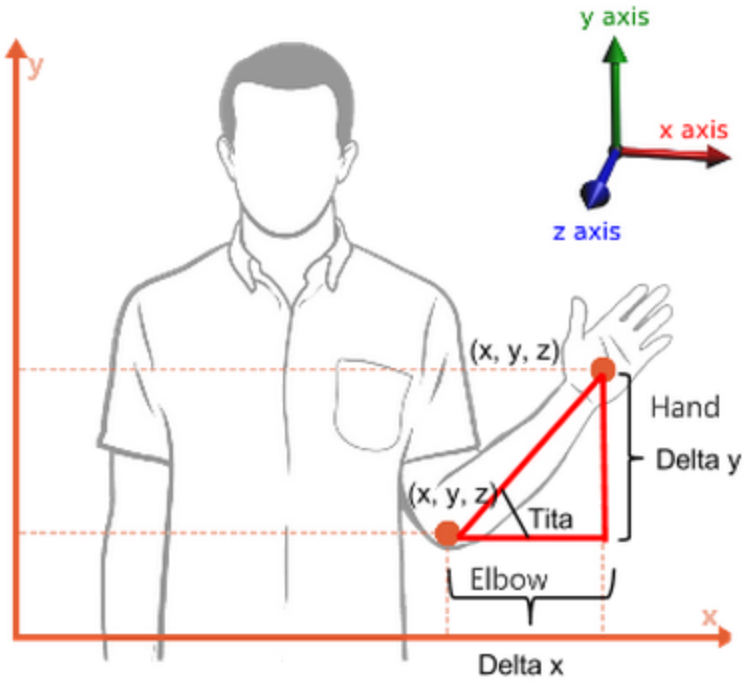

fig. 16 - Captura del ángulo para una articulación

Como se puede observar en la imagen, el eje Z es el que mide la profundidad, que no es parte de la posición por lo que se realiza una proyección en un plano de la posición de cada articulación y se descarta las coordenadas en ese eje. Teniendo en cuenta las coordenadas en los ejes X e Y, se calcula la variación en cada eje y a partir de estas, por medio del uso de trigonometría se calcula el ángulo de dirección.

Por medio del uso del ángulo, se evitan los problemas que podrían surgir de tener distintas personas con distintas alturas, o usuarios que se acercan o alejan del sensor, ya que el ángulo tita no varía en ninguno de estos casos.

El SDK de Kinect en muchos casos envia cuadros con información errónea. Esto es una realidad y ocurre porque el sensor ocasionalmente puede fallar y tomar como parte del cuerpo objetos que se encuentren detrás del usuario, o incluso reflejos de luz. La frecuencia de errores depende de la luz del lugar, la distancia del usuario y el contraste con el fondo entre otras. Ya que el PositionAnalyzer ejecuta acciones en función de las posiciones detectadas, es necesario filtrar estas fallas para evitar ejecutar acciones erróneas. Como la aplicación funciona al mismo rate que el sensor, este módulo recibe updates a un ritmo de

30fps, que es un nivel mucho más elevado que el nivel de interacción del usuario. Esto nos permite crear un promedio de los últimos n ángulos tita detectados, y actuar en función de esta detección, en lugar de cada tita particular. El valor medio nos va a filtrar efectivamente los valores que son errores del sensor, ya que se trata de una medición completamente distinta a las anteriores y se le va a restar importancia en el cálculo del promedio.

El parámetro MediaSize va a definir la cantidad de frames utilizados para el cálculo del promedio de tita. Para MediaSize n, se va a calcular el tita actual y se lo va a promediar con los n-1 últimos valores de tita. Este parámetro nos permite no solo filtrar errores, sino que también nos permite filtrar movimientos involuntarios, por ejemplo si el usuario entra en el sector de un bucket sin querer puede permanecer durante unos pocos frames que dentro del promedio considerando las posiciones anteriores y las posiciones siguientes evita que se ejecute la acción. Sin embargo, a medida que este valor crece, también crece el número de frames que el usuario tiene que permanecer en un sector para ser detectado, y si este valor es elevado puede simular lag desde el punto de vista del usuario.

Es importante resaltar que al utilizar el promedio, al tener una posición detectada como válida, es muy probable que la posición siguiente también resulte detectada. Esto se debe a que si bien es posible que el usuario haya estado durante solamente un frame en el bucket, la media considerando la posición anterior y la siguiente (que para una pasada por la posición va a ser un número un poco menor para la posición anterior y un número un poco mayor para la posición siguiente) probablemente de un valor similar. Esto puede generar que el usuario genere múltiples acciones con solo un frame dentro de la dirección detectada. También es posible que esto lo genere el usuario por cuenta propia, ya que se detectan 30 posiciones por segundo que es un valor mucho mayor a la velocidad a la cual se mueve el usuario generalmente. Por esta razón es que existe un delay entre detecciones de buckets que desactiva el uso para un bucket particular por un intervalo de tiempo luego de una detección, para evitar detectar múltiples veces una unica posicion del usuario.

4.3.7 Comparación de Gestos

Como fue explicado anteriormente, un gesto es considerado una sucesión de skeletons detectados en cada frame. Para la comparación, en cada *update* al recibirse un frame se va a actualizar la ventana que muestra los últimos *n* frames (donde *n* es la cantidad de frames del movimiento contra el que se compara) y se va a realizar la comparación para verificar si se trata del mismo movimiento o no. La comparación consiste en la suma de los errores particulares de cada frame, con lo que se construye un error general del movimiento y si este es menor que el umbral configurado, entonces se considera válido el movimiento. La comparación de cada frame se compone de una suma de las diferencias absolutas al compararse los ángulos generados por cada articulación habilitada.

También existe la posibilidad de realizar la comparación por medio de las posiciones (considerando solamente los valores en los ejes x e y) por medio del método *comparePositions* previa utilización de las funcionalidades de escala (para normalizar los skeletons y evitar las diferencias generadas por distintas alturas o acercamiento / alejamiento al sensor).

5. Interfaz VLC

5.1 Controlador

Para reproducir los distintos sonidos o videos seleccionamos VLC Media Player, por sus avanzadas posibilidades de integración por medio de distintas vías, entre los que ofrece distintos métodos de comunicación con el programa, ejecución de comandos remota y streaming saliente y entrante de video (más información). En primer lugar interactuamos con VLC por medio de la WebAPI pero luego modificamos la implementación para comunicarnos via Remote Control, ya que la WebAPI generaba algunos problemas cuando se efectuaba la ejecución de numerosos comandos de manera encadenada, ocasionados por la latencia introducida. Ver la sección **5.2 Conexión vía WebAPI**.

5.1.1 Interfaces disponibles en VLC:

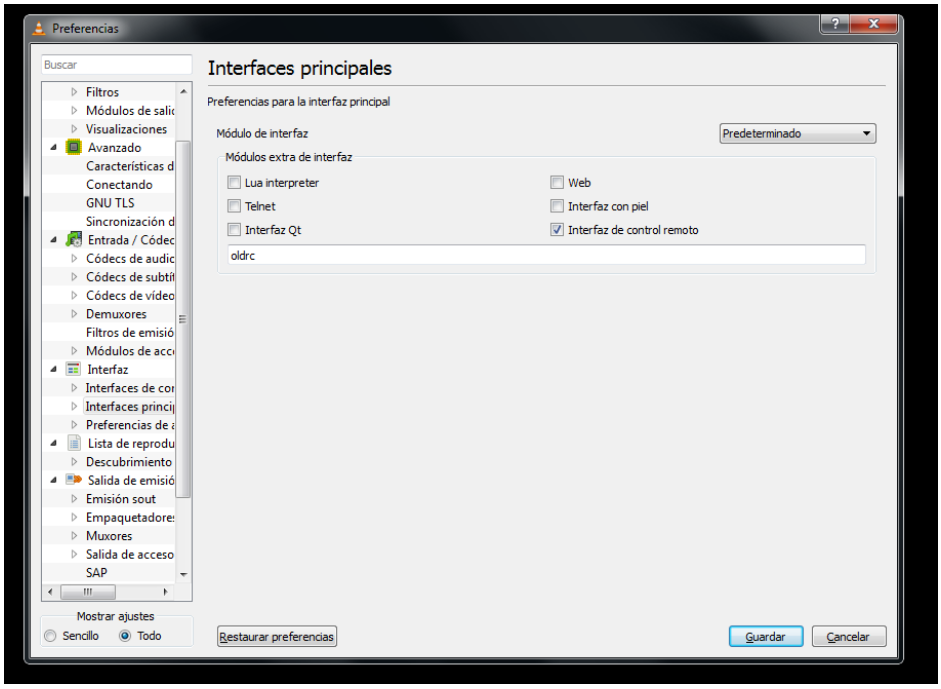

fig. 17 - Listado de posibles interfaces que el VLC nativamente ofrece. *Notar que la que utilizamos es la denominada como Interfaz de Control Remoto*

El proyecto se comunica con VLC por medio de una interfaz llamada VLCController. Esta interfaz define los métodos básicos por medio de los cuales la aplicación va a interactuar con el reproductor. Al estar definida la interfaz, se pudo realizar un cambio en el tipo de comunicación con el reproductor (de WebAPI a RC) en forma totalmente transparente para el resto de la aplicación.

Dentro de la interfaz VLC Controller, encontramos los siguientes métodos implementados:

- shutdown: Desconecta el cliente y lo elimina de la lista de Listeners
- toggleVolume: Alterna entre los valores 1 y 255 del volumen, en función del estado actual
- normal: Setea el valor de ‘Velocidad de Reproducción’ de la pista actual en 1x
- slower: Decrece el valor de ‘Velocidad de Reproducción’ de la pista actual en 1x
- faster: Acelera el valor de ‘Velocidad de Reproducción’ de la pista actual en 1x
- noVolume:Setea el valor del ‘Volumen’ de la pista actual en el valor 1
- fullVolume: Setea el valor del ‘Volumen’ de la pista actual en el valor 255

- togglePlayPause: Alterna entre la reproducción y la pausa de la pista actual
- stop: Detiene la pista y la vuelve al comienzo.
- setup: Mediante una combinación de comandos, no importa cual sea el estado actual de las 6 pistas, las pone en volúmen 255, al inicio y listas para ser reproducidas. Este comando debería ser ejecutado en cada ciclo de la aplicación

Cabe resaltar que hay métodos *toggle*, que pueden mantener un estado interno para conocer qué acción debe realizar, o pueden decidir no mantenerlo y consultar el estado actual al reproductor. En ambos casos, es importante señalar las limitaciones que se presentan al interactuar con un proceso externo, que va a ser modificado por fuera de la aplicación y que puede causar problemas con la representación interna del sistema, si queda de alguna forma desincronizado.

El VLC retorna frente a cada comando enviado, una respuesta/acuse de recibo para indicar que la llamada fue exitosa.

El reproductor VLC tiene comandos que dependen de su estado actual. Dentro de estos comandos se encuentra el caso de stop, que solo realiza la acción en el caso en que se éste en modo play, es decir, reproduciendo el archivo, pero por ejemplo no se ejecuta si la reproducción está pausada. El mismo comportamiento se presenta para el comando de pausa, que si esta pausado sólo puede ser reanudado por el mismo comando “pause” y no por play. Por otro lado, los comandos de velocidad (slower, faster) dependen del estado actual, pero se aplican siempre aumentando o reduciendo la velocidad con respecto a la actual.

5.2 Conexión vía WebAPI

Vlc ofrece una interfaz web que puede ser activada fácilmente y permite interactuar con el reproductor por medio de un navegador o *browser*.

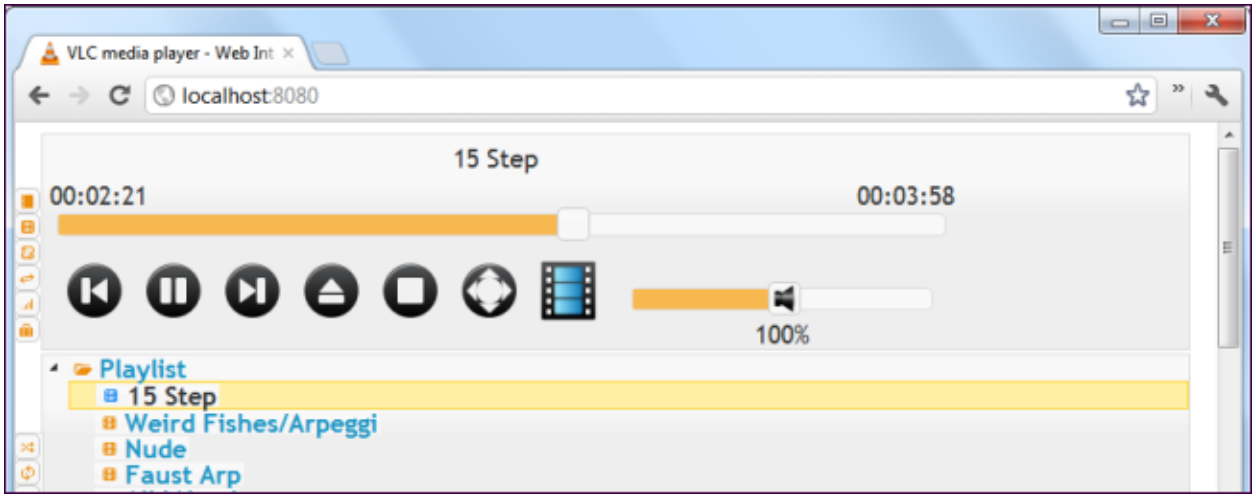

fig. 18 - Interfaz WebAPI

Esta interfaz, permite ejecutar comandos básicos sobre el reproductor, como es el cambio de volumen, o play pause stop. Por defecto la interfaz no esta habilitada, pero se puede configurar fácilmente por medio de la interfaz visual siguiendo este tutorial. Es importante considerar que lo que se genera es un servidor web, y por lo tanto hay que tener en cuenta la configuración del firewall tanto de la pc como de la red para permitirlo. La primera versión del VLCController se implementó por este medio debido a su facilidad de uso, ya que la web interface es fácil de configurar y utilizar, básicamente cada comando de la web se simulaba por medio de un request http a la misma url desde el código. A este request había que agregarle los headers de autenticación (utiliza el header `"Authorization"`), ejecutarlo y esperar la respuesta. Esta implementación fue funcionalmente correcta, sin embargo los request http son reconocidos por su overhead y su ejecución asincrónica, por lo que al intentar realizar muchos request en poco tiempo se terminaban encolando y el tiempo de respuesta aumentaba, probablemente favorecido por el reproductor que limitaba una conexión que realizaba muchos requests. Además, los distintos requests no mantenían una conexión, por lo que resultó en un overhead del protocolo. Por estos motivos, al utilizar seis clientes VLC en simultáneo y ejecutar diversos comandos sobre todos ellos se generaba lag en los reproductores, lo que nos llevó a realizar una implementación más adecuada.

5.3 Conexión vía RC (Remote Control)

El módulo de remote control de VLC permite ejecutar comandos en el reproductor por medio de una conexión a un puerto TCP. Cuando inicializa el reproductor, comienza a escuchar en un puerto determinado por su configuración y permite que otro proceso establezca una

conexión y envía comandos en strings ascii para ejecutarlos en el reproductor. Por el mismo canal luego de la ejecución, la aplicación retorna el resultado del a ejecución, junto con información adicional. En cualquier momento el reproductor puede enviar información sobre su estado, por ejemplo cuando carga un nuevo video, o alguien ejecuta pausa desde la GUI. Los comandos disponibles para ejecutar via Remote Control son similares a los disponibles por WebAPI, el listado completo se puede ver por medio del comando “help” (o por medio de este link).

Para conectarse al puerto uno puede utilizar cualquier cliente TCP, como puede ser telnet, que simplemente con ejecutar telnet ip puerto se conecta y nos abre una terminal por donde se pueden enviar comandos y recibir información. Al igual que en el caso de la WebAPI, es necesario tener en cuenta la configuración del firewall, para permitir conexiones externas al puerto seleccionado para el servidor de VLC.

Remote Control mejora sustancialmente la eficiencia de la comunicación, ya que realiza una primera conexión con su correspondiente *handshake* y una vez establecida se puede seguir utilizando para enviar todos los comandos que se necesiten, cerrando la conexión al finalizar su uso. Si bien esta conexión no genera ningún *overhead* y uno puede enviar comandos muy rápidamente, detectamos que el reproductor tiende a saturarse cuando recibe muchos comandos consecutivos en un corto periodo de tiempo, por lo que decidimos que el VLCController después de ejecutar un comando genere una ventana de 100ms dentro de la cual se ignoran los comandos sucesivos, a fin de evitar este tipo de saturación que podría ocasionalmente generar efectos no deseados.

5.4 Ejecución de las instancias VLC desde consola

```
C:\Program Files (x86)\VideoLAN\VLC>vlc --extraintf rc --rc-host ip_de_red:puerto
```

Es importante resaltar que el VLC escucha en un puerto dentro de la subred definida, por lo que si se utiliza como parametro 127.0.0.1 solo va a ser posible conectarse desde la misma computadora. Es por ello que debe definirse la IP pública de la terminal VLC que se desee ejecutar.
Se pueden encontrar ejemplos de cómo ejecutar la instancia de VLC en la siguiente figura

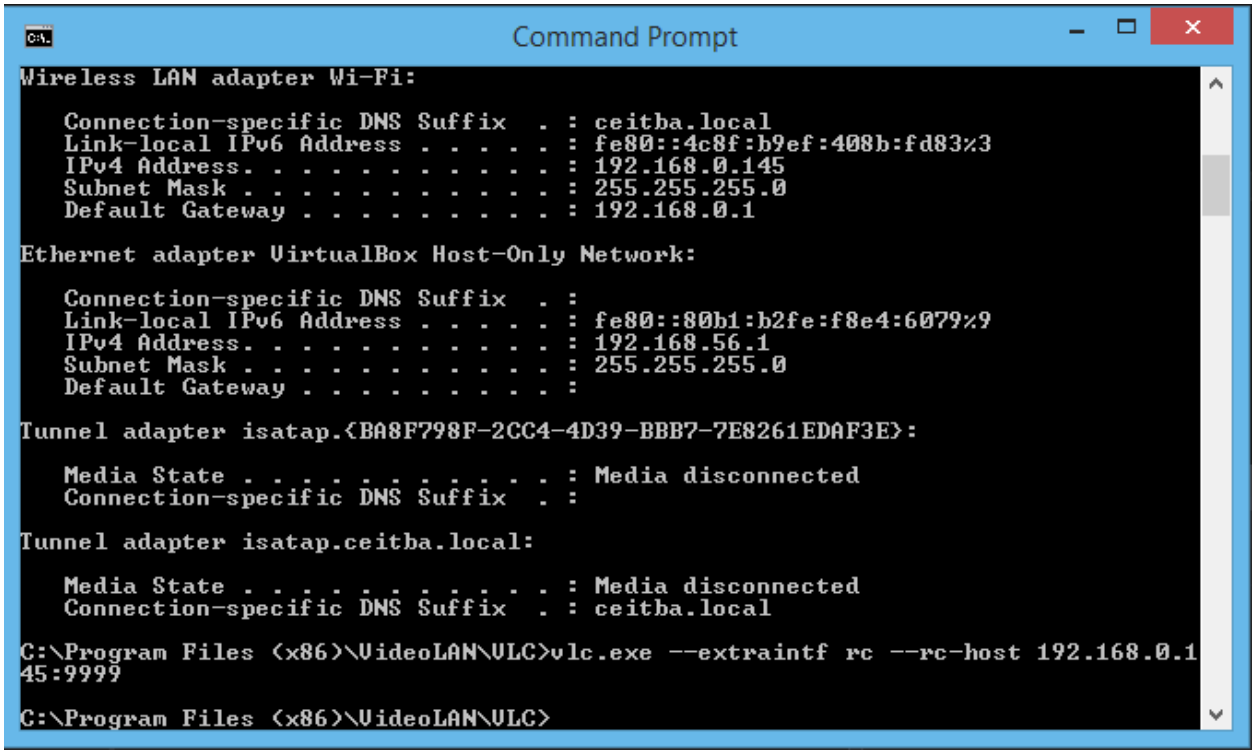

fig. 19 - Ejecución de una terminal VLC

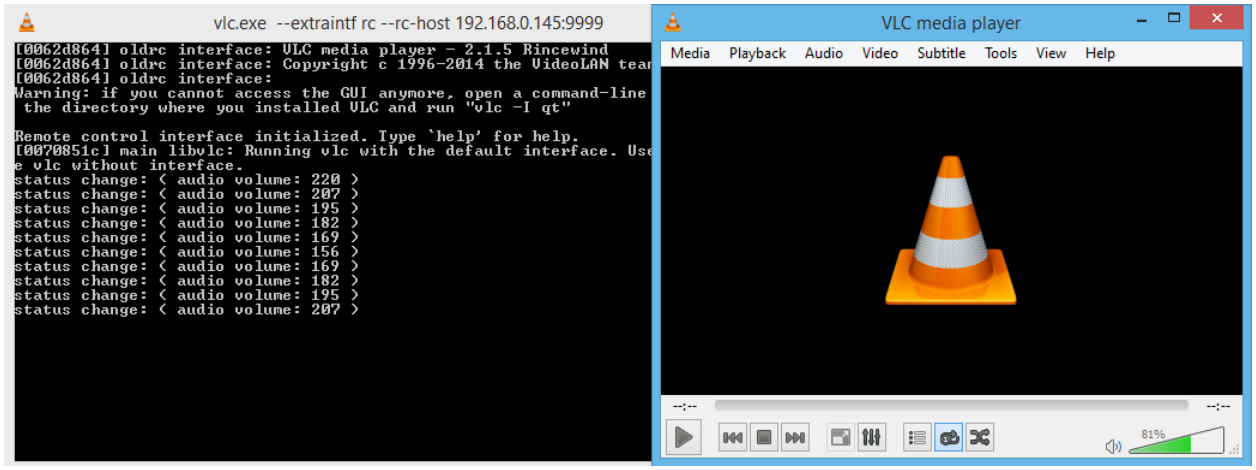

fig. 20 - Visualización de la terminal (comandos ejecutados) asociada a la instancia VLC

donde 192.168.0.145 es la IP pública de la máquina que estará ejecutando la terminal VLC y 9999 es el puerto que expondrá para ser controlado de manera remota.

5.5 Creación de una instancia de VLCController:

Para crear una instancia es necesario proveer un string con la ip y un entero (integer) con el puerto al que se desea conectar. En la creación del objeto se va a establecer la conexión y crear el NetworkStream que se va a utilizar para transmitir los comandos y recibir las actualizaciones del programa. Una vez creado el objeto, en cada ejecución de comando se va a escribir por el stream y leer la respuesta (no bloqueante). Una vez finalizada la utilidad de la instancia, se debe llamar al método shutdown para cerrar correctamente la conexión.

5.6 Hot-swapping

Como se explicó antes, la aplicación realiza interacción con el reproductor VLC por medio de una interfaz llamada VlcController. Esta clase nuclea todo el comportamiento y comunicación con el reproductor. Por cada instancia del reproductor, y hasta un máximo de 6 instancias en simultáneo se va a crear un VlcController para permitir comunicarse. Al haber distintos módulos que pueden comunicarse con el reproductor, es necesario que cada uno de ellos conozca las instancias de VlcController. No es performante crear una instancia del controlador por cada módulo distinto que interactúe, ya que se estarían creando múltiples conexiones para conectarse con un único reproductor, que llevaría un *overhead* tanto de comunicación como de procesamiento. Para evitar esto, las instancias se crean y guardan en un único array que es accedido por los distintos módulos por medio de llamadas y vistas. Esto nos permite compartir la misma instancia del controlador entre los distintos módulos, que van a llevar una vista estática de ese array, conociendo su estructura en real-time pero sin poder modificarla (asegurado por medio del uso de un IReadOnlyList<VlcController>). Además del beneficio en eficiencia, esto nos permite realizar la estructura del hot-swapping de reproductores. Basta con que la interfaz visual deshabilite una posición en ese array para que se vea reflejada en todos los módulos que interactuar con el reproductor y comiencen a ignorarlo. De la misma forma al agregar una nueva conexión, se fija la nueva instancia de VlcController y se asigna en el array, por lo tanto, cualquier módulo que en adelante pregunte por esa posición del array va a ver el nuevo controlador y va a poder interactuar con el.

Esta arquitectura de concentración y manejo de los controladores en un solo sector del codigo es lo que nos permite modificar en cualquier momento de la ejecución los controladores y nos permite tener un seguimiento exhaustivo de los recursos utilizados y evitar un *leak* de conexiones.

6. Interfaz de Usuario

La interfaz de usuario, realizada en WPF, provee al usuario de un menú de configuración que le permite ajustar los diversos parámetros del programa. En el centro de la pantalla observamos la representación en esqueleto, y mediante puntos y trazos, de la posición en tiempo real del usuario.
Desplegados en cuadrados negros, se visualizan los 6 buckets, cuyos estados iniciales (rojo) indican que se encuentran desconectados.
El círculo en la esquina superior derecha indica el reconocimiento de los gestos de voz, y el modo en el que actualmente se encuentra el programa (Buckets - Gesture Control)
A la izquierda se encuentran los comandos que le permiten al usuario de la aplicación poder grabar su propio gesto, previsualizarlo y persistirlo (Start recording - Play - Save)

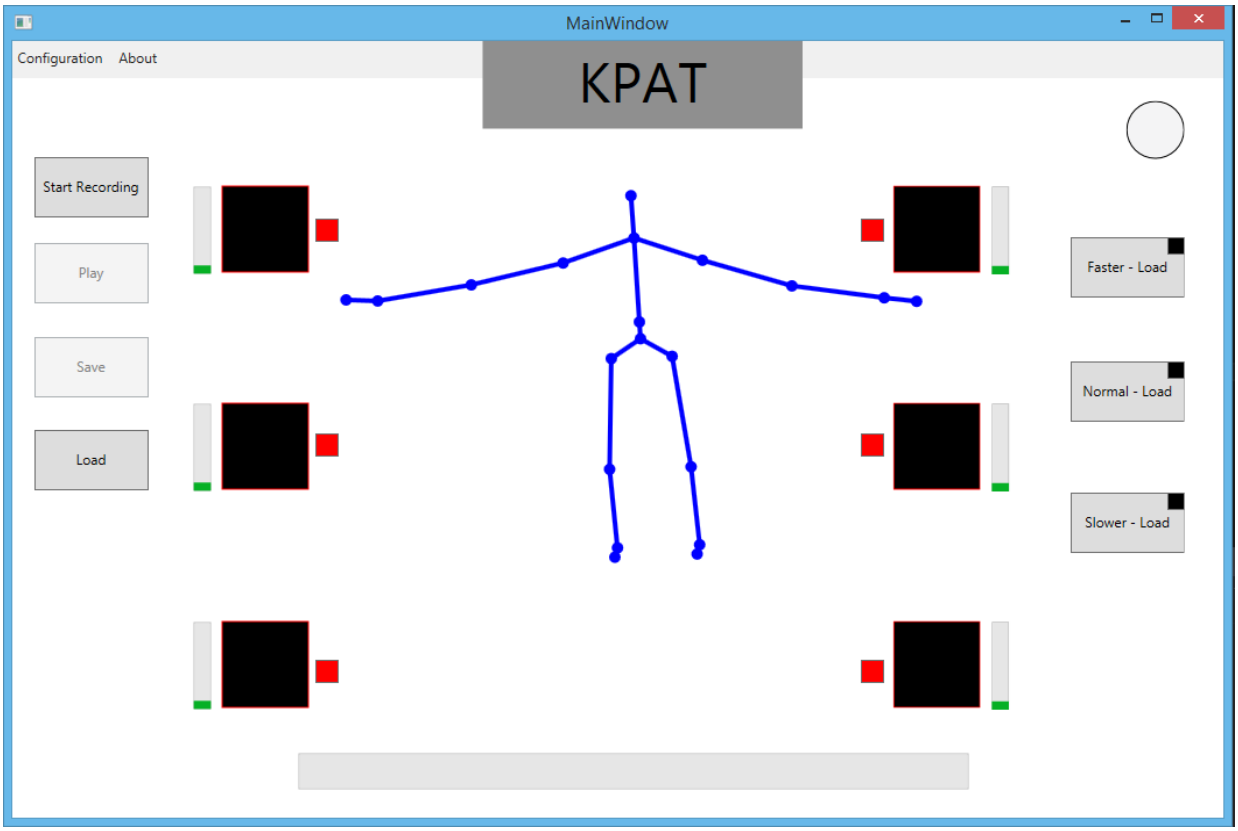

fig. 21 - Interfaz visual del aplicativo KPAT

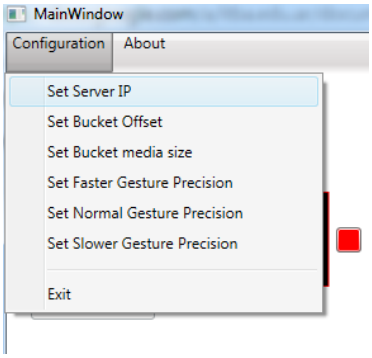

fig. 22 - Menú de configuración de KPAT

A continuación se presenta el menú de configuración de KPAT cuyos comandos pertenecen al siguiente listado.

ServerIP: Representa la dirección IP de la instancia KinectServer. Por default, el mismo asume que se encuentra ejecutando en la misma máquina, bajo la IP 127.0.0.1 (Localhost)

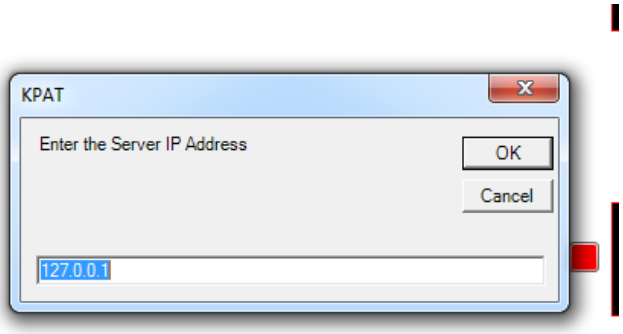

fig. 23 - Prompt para ingresar la IP del servidor

Bucket Offset: Este parámetro determina el offset, medido en grados, definido para cada región de reconocimiento, como se explica en la sección **7.1 Calibración** (Por defecto se asume en 12º)

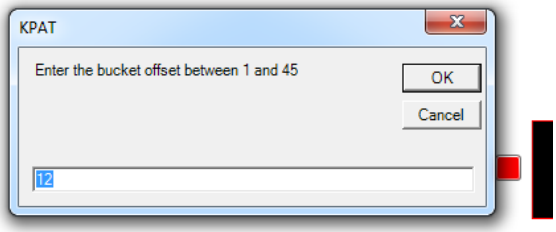

fig. 24 - Prompt para ingresar el offset asociado a cada región de reconocimiento de Buckets

Bucket Media Size: Determina sobre cuántas muestras se realizará el promedio para determinar la equivalencia entre dos series de fotogramas (Skeletons), tal como se explica en la sección
4.3.6 Comparación de posiciones. Su valor por defecto es de 10 muestras, y surge del análisis empírico como se detalló en la sección **7. Análisis y Resultados**

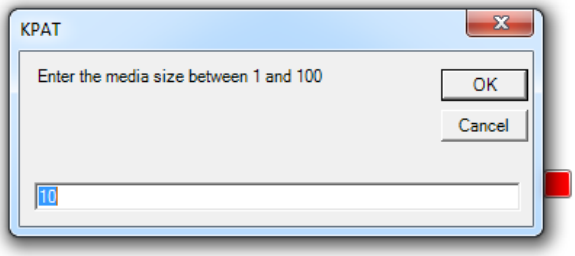

fig. 25 - Prompt para ingresar el tamaño de la media para el promedio de buckets

Slower/Normal/Faster Gesture Precision: Determina la precisión (umbral) con la que cada movimiento será detectado. Cada gesto tendrá una precisión determinada, y podrá ser modificada independientemente

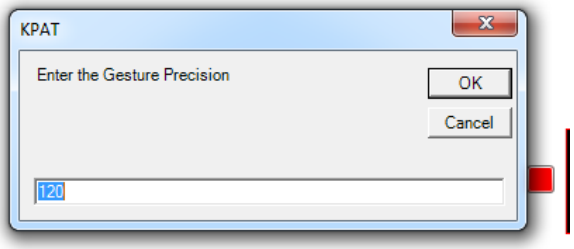

fig. 26 - Prompt para ingresar el la precisión de alguno de los tres gestos definidos.

En la parte superior derecha encontraremos el indicador de estado del modo de la aplicación, en función del color que tome, indicará el modo en el que se encuentra tal como indica la siguiente figura.

fig. 27 - Modos de la aplicación

Blanco: Reconocimiento inicialmente desactivado
(Es necesario realizar el comando SET-UP)

Amarillo: Reconocimiento inhabilitado (buckets)

Verde: Reconocimiento activado

En cada bucket se puede observar los siguientes colores que indican a su vez, el estado en el que se encuentra cada uno de los mismos.

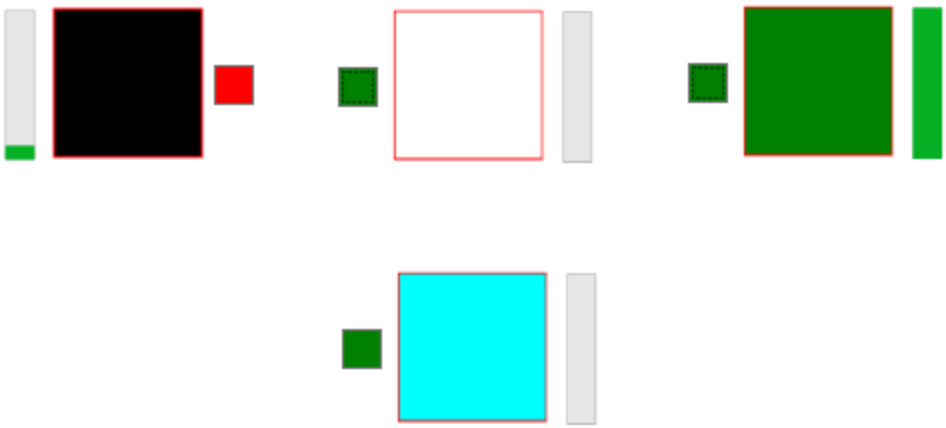

fig. 28 - Estados de los buckets

Negro: Bucket inactivo (Es necesario configurar su IP)

Blanco: Bucket con volúmen en valor 1 (Mute)

Verde: Bucket con volúmen en valor 255 (Máximo)

Celeste: Bucket actualmente seleccionado (Feedback al usuario)

6.1 Comandos de Voz

Para la interpretación de la voz, se hace uso de la composición de una librería de reconocimiento de voz. Hay que indicar que la misma utiliza una librería nativa de idioma del sistema operativo y hemos probado la funcionalidad con la librería en-US, para detección de comandos en el idioma inglés.

Para la manipulación de la aplicación se define la siguiente lista de comandos aceptados, cuyas acciones son detalladas.

SetUp: Mediante una combinación de comandos, no importa cual sea el estado actual de las 6 pistas, las pone en volúmen 255, al inicio y listas para ser reproducidas. Este comando debería ser ejecutado en cada ciclo de la aplicación

Buckets: Ingresa en el modo Gesture Control, en donde el visor se coloca en color amarillo, y el reconocimiento de posiciones está desactivado, para poder ejecutar los comandos a través de gestos

Play/Pause: Reproduce o pausa la reproducción actual

Nota: Cada comando de voz es, en primera instancia, detectado, luego hipotetizado y luego inferido, dependiendo si el nivel de reconocimiento supera el valor empírico 'confidence' cuantificado en 0.80

7. Análisis y Evaluación

7.1 Calibración

Con el fin de obtener la mejor experiencia al utilizar la aplicación, se tuvieron que realizar numerosas pruebas para ajustar y calibrar las zonas efectivas en donde los movimientos son detectados. Dentro de estas pruebas, también nace la necesidad de crear una ventana entre cada ejecución de un comando, debido a que los mismos se encadenan, si se mantiene la misma posición a través del tiempo. Asimismo también fue necesario calibrar el umbral de los gestos para los cuales los mismos son detectados.

Para calibrar las zonas efectivas, se desarrolló un pequeño programa que genera trazos de color, a través del ángulo formado por la extensión del brazo y la horizontal. El efecto final era una división del espacio en distintos colores, según se prefijaban intervalos respectivos. El mismo puede ser observado en la siguiente figura.

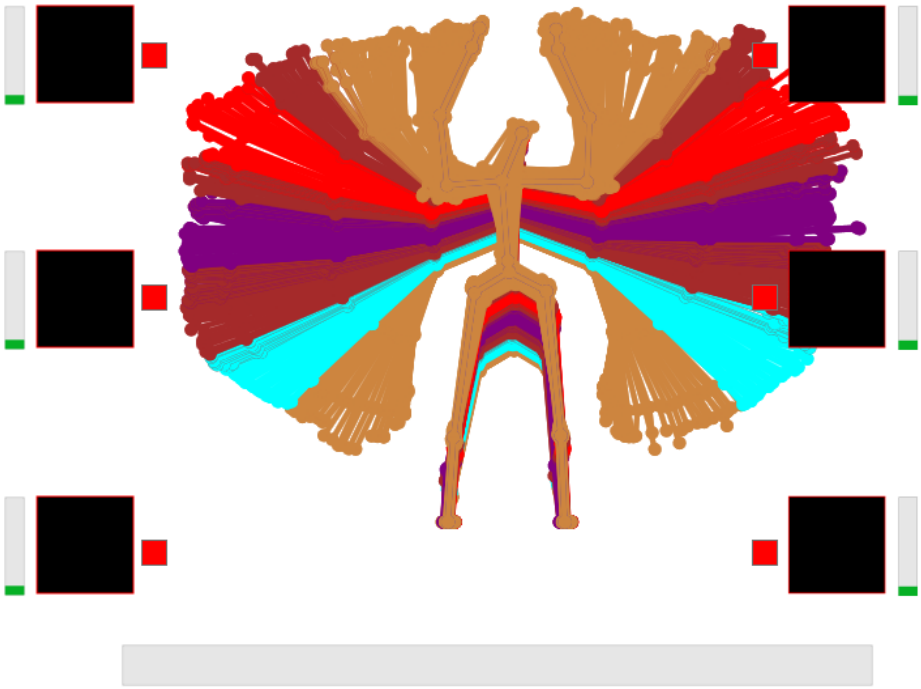

fig. 29 - Esquema que muestra cómo se divide el espacio en distintas zonas de reconocimiento

El esquema se explica bajo el siguiente código de colores

Marrón: Zona de detección inválida, ya que se realiza una comprobación de la posición del codo respecto de la muñeca, para evitar falsos positivos, ante una incorrecta ubicación de dichas articulaciones.
Celeste. Zona de detección Inferior +- Offset
Bordo: Espacio entre buckets que representa una zona neutral de detección
Violeta. Zona de detección Media +- Offset
Rojo. Zona de detección superior +- offset

Nota: Para eliminar falsos positivos, como se indicó, se realiza la comparación de la posición de la Joint (Articulación) Elbow menos la posición de la joint Wrist , y si la misma es menor a un umbral determinado (Empíricamente se definió en < 0.1) el movimiento se considera inválido de ser detectado.

De esta manera, mediante la configuración de los intervalos angulares, se podría dividir el espacio de manera equitativa y efectiva. El efecto fue poder dividir las zonas efectivas de reconocimiento y asociar cada zona a un ‘bucket’ en particular. A su vez, se define un offset

general, el cual, definido el cero para cada zona, generan finalmente la zona final de reconocimiento siguiendo la fórmula: cero +- offset. Así se divide cada espacio (espacio izquierdo asociado al brazo izquierdo, y espacio derecho asociado al brazo derecho) en 3 zonas o umbrales. El offset fue definido en 12º, generando intervalos de 24º de extensión, para generar la mejor experiencia de uso.

Luego de las experiencias realizadas, las mismas quedaron definidas en:

Zona Inferior: 38º +- 12º
Zona Media: 90º +- 12º
Zona Superior: 125 +- 12º

Definidas las zonas, lo siguiente a calibrar fue el tiempo de retardo entre cada movimiento reconocido. Para una experiencia que resulte usable, y no genere encadenamiento de comandos indeseables, se definió entre 1s (secsDelay) entre reconocimiento y reconocimiento.

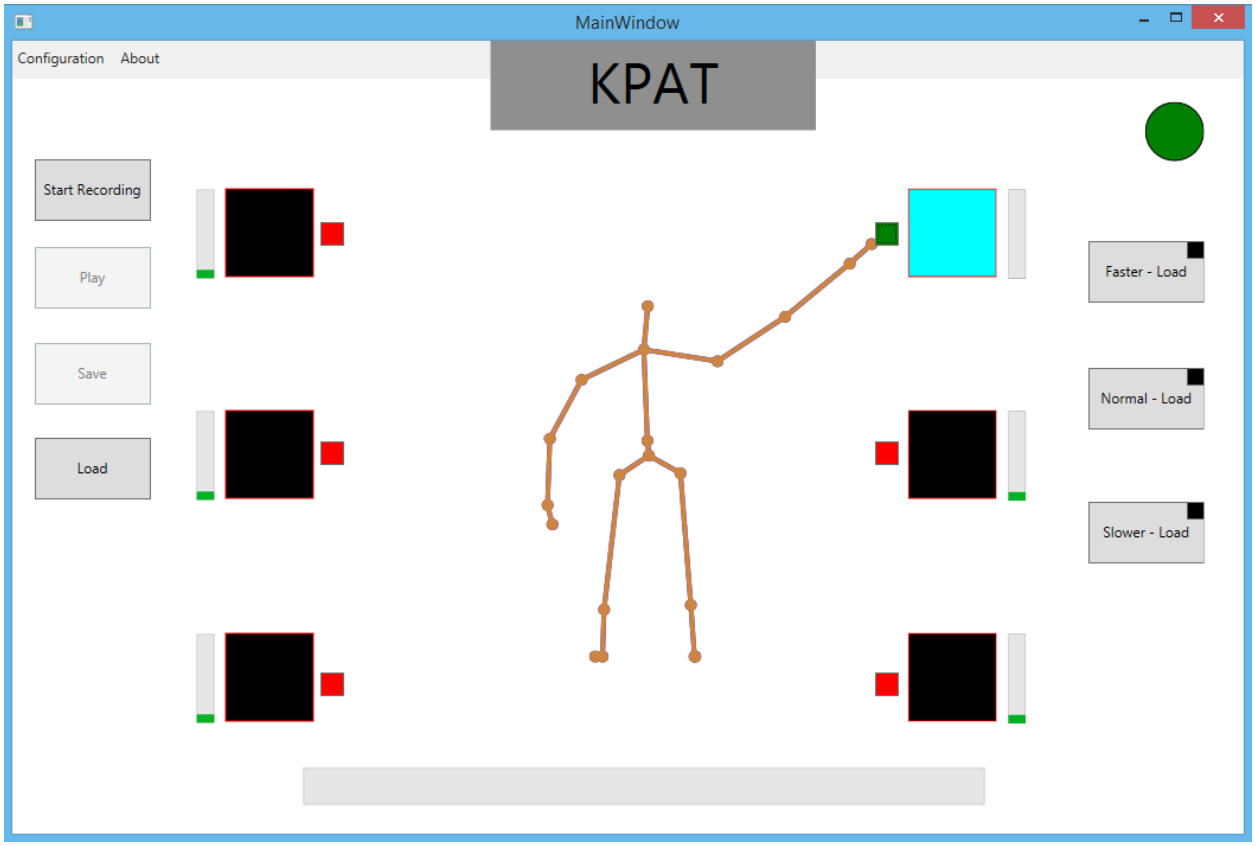

fig. 30 - Reconocimiento efectivo del bucket superior derecho

Lo siguiente fue definir los gestos predefinidos para la demo, e incorporarlos en la entrega final.
Para ello se evaluó la incorporación de numerosos gestos, que fueran sencillos de lograr y replicar. Los tres (3) gestos incorporados se denominan final.slower, final.normal, final.faster, los cuales controlan la velocidad de reproducción de las pistas del VLC Media Player.

Para cada gesto de control de velocidad, se añadió la posibilidad de definir el umbral de reconocimiento.

fig. 31 - Reconocimiento nulo del gesto efectuado

fig. 32 - Reconocimiento parcial (No supera el umbral de detección) de un gesto

fig. 33 - Reconocimiento de un gesto

Esta propiedad nace de las diversas dificultades de poder reproducir la mímica de cada gesto.
En base a evaluaciones empíricas, nosotros como autores sugerimos el siguiente set de precisiones recomendadas

Precisión Faster: 105
Precisión Normal: 60
Precisión Slower: 125

En base a estas precisiones, se tiene un sólido reconocimiento de los gestos, cuando son correctamente efectuados.

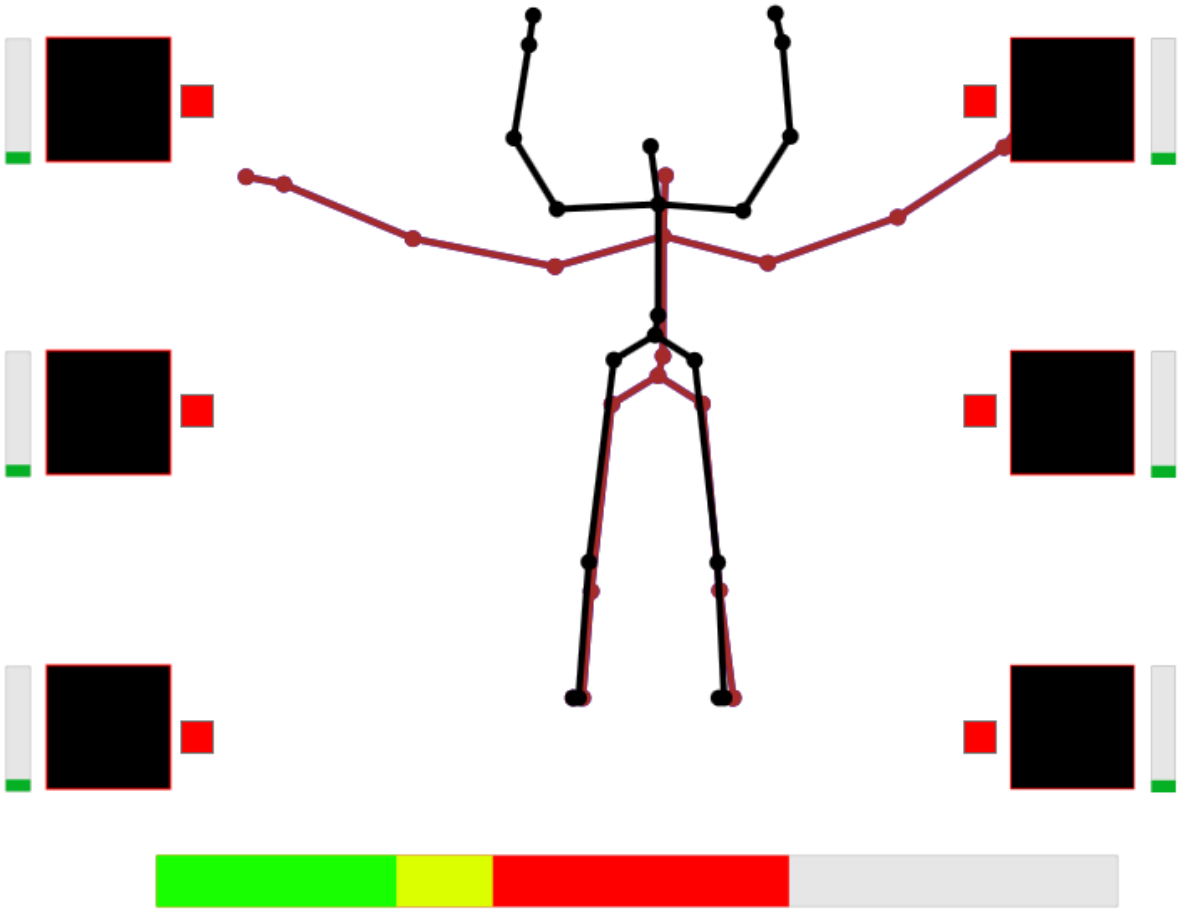

fig. 32 - Reconocimiento de un gesto (Preview)

7.2 Plan de Pruebas

Para poder corroborar que los parámetros fueron bien escogidos, tuvimos que acudir a un plan de pruebas empírico. Para ello elaboramos un set de 100 repeticiones, en las cuales la intención era a partir de un movimiento natural de extensión del brazo, y a una velocidad rápida, lograr habilitar el bucket deseado.
Se definió una aceptación esperada del 97%, es decir, 97 de cada 100 repeticiones naturales, el bucket deseado debería haber sido seleccionado.

Para poder corroborar la precisión de cada gesto, se obró de manera similar, generando 30 iteraciones para cada gesto, y definiendo una aceptación del 90%, es decir, 27 de cada 30 gestos, realizados por el creador del mismo, deberían ser reconocidas por la aplicación.

7.3 Resultados

Con el objeto de sumarizar todo lo explicado en este informe, y debido a que nuestra proyecto desarrolla una aplicación de carácter práctica, se realiza un video Demostración (Demo) del funcionamiento de la misma.

El mismo puede ser encontrado en Google Drive, bajo los siguientes enlaces de carácter públicos:

Parte I - Configuración:
https://drive.google.com/file/d/0B3DlxzykRjuFTXJpazNvMnZHdkU/view?usp=sharing

Parte II - Reconocimiento de Posiciones:
https://drive.google.com/file/d/0B3DlxzykRjuFT2x5d3RDdlhCc2c/view?usp=sharing

Parte III - Reconocimiento Gestual (1):
https://drive.google.com/file/d/0B3DlxzykRjuFUnEtTkRrdVVvNVU/view?usp=sharing

Parte IV - Reconocimiento Gestual (2):
https://drive.google.com/file/d/0B3DlxzykRjuFcU9lOFBkVENRVVk/view?usp=sharing

8. Conclusiones y Trabajo Futuro

La principal motivación de este proyecto es poder construir una aplicación interactiva que desencadene acciones a partir de gestos que pueden ser customizados por el usuario. Para esto, se construyó una aplicación distribuida, que a partir de un sensor Kinect, se despliega al usuario la información sensada, y el mismo puede interactuar para utilizar una interacción a un VLC Media Player. La interfaz en Unity fue una opción inmediata, como integración para poder desplegar la vista al usuario, pero la misma presentó complicaciones al ser implementada, tal como fueron descritas en el desarrollo de este informe, por lo que se tuvo que realizar la misma en WPF.

Creemos que a partir de nuestra aplicación, se pueden construir otras que funcionen como promotores de aprendizaje. Por ejemplo, para ejercitar el oído musical, se podría construir una aplicación que desactive azarosamente alguna de las pistas, y el usuario deba indicar con sus brazos cual cree que fue la pista que dejó de sonar. Cuanto más rápido lo hace, mayor puntaje obtiene. De esta forma, el discernimiento musical se estaría entrenando, a partir de una aplicación que combina multimedia con interacción gestual.

Como trabajo futuro o posibles extensiones sobre el proyecto presentado, se pueden considerar las siguientes sugerencias:

- Retomar el desarrollo de una interfaz 3D en Unity para permitir un gran salto de calidad en la interfaz visual.
- Realizar streaming de video con las instancias de VLC para reflejar en la UI lo que está transmitiendo cada terminal en real-time.
- Efectuar un análisis más profundo de los distintos algoritmos para perfeccionar la detección de posiciones y movimientos y reducir el error de los algoritmos utilizados.
- Incorporar módulos a la infraestructura para permitir realizar otras acciones en base a la información recibida.
- Utilizar el sensor de profundidad brindado por el SDK de Kinect para mejorar la detección.
- Realizar alguna aplicación que utilice la estructura pero que sea multiusuario.

9. Bibliografía

1. Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. 1985. Direct manipulation interfaces. Hum.-Comput. Interact. 1, 4 (December 1985), 311-338.

2. http://e-archivo.uc3m.es/bitstream/handle/10016/16895/TFG_Juan_Toribios_Blazquez.pdf?sequence=1

3. Salvador, S., Chan, P.. FastDTW: Toward Accurate Dynamic Time Warping in Linear Time And Space.

4. Müller, M. 2007. Information Retrieval for Music and Motion, Springer. 10.

5. GangOfFour, 1994. Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley Professional.

6. http://www.mblondel.org/journal/2009/08/31/dynamic-time-warping-theory/

7. http://www.saaei.org/edicion2011/papers/SE1/SE106.pdf

8. http://gdsproc.com/congreso/Sistema%20de%20Reconocimiento%20de%20Rostros.pdf
9. http://channel9.msdn.com/coding4fun/kinect/KinectSDK--Unity3D-Interface-v5-now-Kinect-for-Windows-SDK-v1-compatible
10. https://github.com/adevine1618/KinectSDK-Unity3D_Interface_Plugin

11. https://social.msdn.microsoft.com/Forums/en-US/749e47fc-8e7b-4286-b095-72c7676ecaaf/unity-jointorientations-to-bone-mapping-not-quite-right?forum=kinectv2sdk
12. https://social.msdn.microsoft.com/Forums/en-US/4a428391-82df-445a-a867-557f284bd4b1/dynamic-time-warping-to-recognize-gestures?forum=kinectsdk

13. https://gist.github.com/socrateslee/1966342

14. https://github.com/doblak/ndtw

15. http://pterneas.com/2014/01/27/implementing-kinect-gestures/

16. https://github.com/lightbuzz/vitruvius

17. http://www.franklins.net/gesturepak.aspx

18. http://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles

19. https://social.msdn.microsoft.com/Forums/en-US/31c9aff6-7dab-433d-9af9-59942dfd3d69/kinect-v20-preview-sdk-jointorientation-vs-boneorientation?forum=kinectv2sdk

20. https://social.msdn.microsoft.com/Forums/en-US/749e47fc-8e7b-4286-b095-72c7676ecaaf/unity-jointorientations-to-bone-mapping-not-quite-right?forum=kinectv2sdk