Trabajo Fin de Grado
Ingeniería en Tecnologías Industriales

Control de Robot Animatrónico mediante Visión Artificial

Autora: Irene Luque Martínez
Tutor: José María Maestre Torreblanca

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Grado
Ingeniería en Tecnologías Industriales

Control de Robot Animatrónico mediante Visión Artificial

Autora:
Irene Luque Martínez

Tutor:
José María Maestre Torreblanca
Profesor titular

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Trabajo Fin de Grado: Control de Robot Animatrónico mediante Visión Artificial

Autora: Irene Luque Martínez

Tutor: José María Maestre Torreblanca

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

A Paco y Malena

Agradecimientos

En primer lugar agradecer a Pepe, Coro y Dani, por el tiempo y el interés, por los consejos, por las ganas y sobre todo por brindarme la oportunidad de llevar a cabo este proyecto que llevaba un tiempo forjándose en sus cabezas. También como empresa a *Puppeteer Technologies*, por abrirnos sus puertas y confiar en nosotros su robot y su trabajo desde el primer momento.

A mi padre, mi madre y mi herma por ser mis pilares desde el principio; literalmente no sería nada sin vosotros. Y a las que habéis llegado un poco después: gracias a mis compañeras por convertirse en amigas, Carla, Marisa; gracias a Marta y María por aguantarme desde el día uno; gracias Manu por tu inagotable paciencia; y por supuesto gracias a todos los demás que habéis formado parte de mi vida a lo largo de este camino.

Irene Luque Martínez

Grado de Ingeniería en Tecnologías Industriales

Sevilla, 2020

Resumen

El siguiente proyecto trata el control de un robot animatrónico basado en la grabación de los movimientos faciales de un actor. Para ello, la cara del robot está provista de ocho servomotores que lo dotan de capacidad de movimiento en ojos, cejas, boca y cabeza. La finalidad del proyecto es que nuestro robot pueda imitar los movimientos y expresiones faciales de un actor a tiempo real, creando así un modelo animado que podría ser usado en proyectos para cine.

Para llevar a cabo el proyecto se han desarrollado códigos de grabación, análisis y procesamiento de imágenes a tiempo real en Matlab. A través del protocolo OSC se creará una red de transmisión de la información, de manera que la versión digital del robot en Maya reproducirá el movimiento originalmente realizado por el actor, dando vida así a nuestro personaje digital animado.

Abstract

The following Project deals with the control of an animatronic robot, based on the recording of an actor's facial movements. In order to achieve this, the robot's face is equipped with eight servomotors that provide it the ability to move its eyes, eyebrows, mouth and head. The Project proposal is that our robot can imitate the movements and facial expressions of an actor in real-time, creating this way an animated model that could be used in film Projects.

In order to carry out our idea, we have developed codes in Matlab that record, analyze and process images in real-time. Through the OSC protocol, an information transmission network will be created so that the digital version of the robot in Maya is able to reproduce the movement originally made by the actor, giving life to our animated digital character.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

Mocap	*Motion Capture* (Captura de Movimiento)
POO	Programación Orientada a Objetos
SDK	*Software Development Kit* (Kit de Desarrollo Software)
API	*Application Programing Interface* (Interfaz de Programación de Aplicaciones)
NUI	*Natural User Interface* (Interfaz Natural de Usuario)
TFG	Trabajo de Fin de Grado
AU	*Animation Unit*
MEL	*Maya Embedded Language*
Fps	*Frames per second*

1 INTRODUCCIÓN

LA Visión Artificial es una disciplina científica de la que mucho se está oyendo hablar en los últimos tiempos. Basta con hacer una rápida investigación para darse cuenta de la importancia que está adquiriendo esta ciencia y de la cantidad y diversidad de aplicaciones que ya tiene en nuestra vida diaria.

Si tuviéramos que definir brevemente en qué consiste esta materia, podríamos decir que se trata de la combinación de conceptos y técnicas para adquirir, procesar, analizar y comprender las imágenes del mundo real con el fin de producir información numérica o simbólica que pueda ser tratada por un ordenador (1).

Lo cierto es que la visión artificial está cada día más integrada en nuestras actividades rutinarias. Estamos haciendo uso de ella desde la descarga de la carta de un restaurante a través de un código QR hasta la detección de fallos en un laboratorio a través de sensores de visión, pasando por su aplicación en metrología, la detección de intrusos o el modelado digital de objetos o ambientes. Entre esta inmensidad de aplicaciones prácticas es donde se está haciendo cada día un hueco más grande su uso para el reconocimiento facial y la creación y movimiento de personajes 3D; es decir, para la animatrónica.

La Animatrónica es una técnica que trabaja en la simulación digital del movimiento de seres vivos (2). La principal utilidad de la animatrónica se da en el campo de la cinematografía y los efectos especiales, y a día de hoy la demanda de este tipo de recreaciones virtuales es cada vez más potente. Los videojuegos, el gaming, la realidad virtual y, por supuesto, las películas de animación, convierten a esta ciencia en un motor de movimiento cultural y económico alrededor de todo el mundo.

Es este precisamente el objeto último de mi Trabajo de Fin de Grado: desarrollar el procesamiento completo que hay detrás del movimiento de un personaje en una película de animación. Para ello hemos tenido la gran suerte de poder colaborar con *Puppeteer Technologies*, una empresa gaditana dedicada a la Animatrónica para cine que estuvo interesada en este proyecto desde el principio y que nos facilitó su software y hardware para poder trabajar.

La Captura de Movimiento es la práctica de la animatrónica en la que nos hemos basado para nuestro proyecto. Se dedica a la grabación del movimiento de actores y animales vivos y el traslado de dicho movimiento a un modelo digital creado a través de imágenes por computadora. Es conocida como 'Mocap', abreviación del inglés 'Motion Capture', y su principal aplicación está en la industria del cine de fantasía o ciencia ficción, en la industria de los videojuegos o incluso en los deportes, con fines médicos (3).

Se podría decir que existen dos sistemas de captura de movimiento bien diferenciados: uno óptico y otro electromagnético y mecánico. El primero de ellos es el más utilizado por la industria cinematográfica y del entretenimiento, siendo así el que nosotros hemos desarrollado en nuestro proyecto (4).

1.1 Motivaciones

La industria cinematográfica comenzó a utilizar la técnica del *Mocap* el año 2000 con la película *Simbad*: fue el primer largometraje de animación por computadora creado exclusivamente mediante captura de movimiento. Después de esto comenzó a normalizarse y pudimos empezar a observar esta disciplina en grandes personajes como Gollum (trilogía *El Señor de los Anillos*, 2001-2003), el simio King-Kong (película *King Kong*, 2005) o Hulk (*The Avengers*, 2012), entre otros muchos más (3).

Desde que el *Motion Capture* pasó a ser una técnica medianamente accesible en las empresas de cine, este mercado no ha hecho más que crecer. En 2019, la industria global de animación registró un valor de mercado de aproximadamente 264.000 millones de dólares, y cada año se prevé un aumento considerable de esta cifra (5).

Puppeteer Technologies es una de las múltiples empresas que se dedican a este mundo y, dado que el registro digital de movimientos faciales que he desarrollado en este proyecto es una técnica que interesa mucho de cara a la industria de la Animatrónica para cine, ambas partes decidimos colaborar juntas en este proyecto. Esta empresa se dedica a desarrollar el middleware necesario para controlar robots, deforma que nos facilitaron un robot físico creado por ellos con el que ya han participado en diversos trabajos, además de su desarrollo software: la aplicación *Puppeteer* y la versión digital del robot en *Maya*.

Figura 1-1.Visualizado de la versión física y digital del robot de Puppeteer (6)

Los proyectos en los que han participado los creadores de *Puppeteer* no han sido pocos. Daniel Quintero es el socio fundador de la empresa. Javier Coronilla, el Animatronic Designer que hay detrás de nuestro robot, ha trabajado en películas como 'Victor Frankenstein', de Paul Mc Guigan; tres episodios diferentes de la saga 'Star Wars' ('Star Wars: Episodio VII – El despertar de la Fuerza', de J.J. Abrams; 'Rogue One: Una historia de Star Wars', de Gareth Edwards; 'Star Wars: Episodio VIII – Los últimos Jedi', de Rian Johnson); 'Un monstruo viene a verme', de Juan Antonio Bayona; o 'El regreso de Mary Poppins', de Rob Marshall. También ha creado criaturas para series de televisión tan populares como 'Doctor Who' (7).

El robot con el que hemos trabajado en este proyecto, que ha sido diseñado y construido por ellos, está dotado de distintos servomotores que le conceden un gran rango de movimiento en todas las facciones de la cara y las articulaciones del cuerpo. Nuestro proyecto está centrado en la captura y procesamiento de imágenes faciales ('Facial Mocap'), por lo que hemos trabajado específicamente con la cabeza de dicho robot.

El modo usual de proceder en la Captura de Movimiento consiste en fijar marcas al sujeto cuyo movimiento se va a analizar y hacer un rastreo de la posición de estas marcas para registrar los movimientos. Las marcas pueden posicionarse a lo largo del cuerpo, si queremos crear un personaje de cuerpo entero, o en distintos puntos de la cara si lo que buscamos es un rastreo facial ('Facial Mocap').

Figura 1-2. Ejemplo de marcas corporales y faciales para Motion Capture (4)

No obstante, pese a que la localización de marcas faciales facilita el rastreo de movimiento, el sistema que he desarrollado es capaz de analizar el movimiento de las facciones de la cara sin necesidad de fijar marcas en ella, lo cual supone un gran avance en este mundo, ya que libera al actor de la necesidad de cargar con marcas alrededor de cada miembro de su cara, que limitan su capacidad de movimiento y su expresividad.

Este TFG proporciona una propuesta innovadora en lo que se refiere al 'Facial Mocap'. Como veremos más adelante, partiendo de unas bases sólidas he desarrollado un software genuino que, aún con las limitaciones que se han teniendo dado el contexto físico y temporal del trabajo, podría llegar a tener aplicaciones comerciales en cine y animación, ya que trata un tema potente y con mucha proyección industrial de cara al futuro.

1.2 Planteamiento

Teniendo en cuenta lo que ya sabemos sobre Animatrónica y Captura de Movimiento, y con el objetivo de tener una idea clara y esquematizada de cómo ha sido el proceso de creación de nuestro personaje animado, vamos a concluir con la Introducción haciendo un breve salto explicativo por los cuatro pasos esenciales que hemos tenido que desarrollar a lo largo del proyecto.

Ya hemos visto que todo movimiento animado comienza con el registro y procesamiento del movimiento de un actor de carne y hueso. Para ello primeramente es necesario recopilar información muy específica sobre las imágenes que se están capturando. Precisamos, por ejemplo, de información detallada sobre el movimiento, sobre la profundidad o sobre la radiación infrarroja de la imagen, para poder así crear un modelo virtual del actor con el que estamos trabajando. En el caso de este Proyecto nos hemos hecho con una cámara *Kinectv2*, dispositivo asociado a la consola *Xbox One* que salió al mercado a finales del año 2010.

Tras un análisis pormenorizado sobre las cualidades de grabación de distintos tipos de dispositivos, del que hablaremos más adelante, nos dimos cuenta de que la cámara *Kinectv2* cumple con muchas cualidades interesantísimas para el procesamiento de imágenes que estábamos buscando, y que la hacen destacar frente al resto de dispositivos que teníamos a nuestro alcance.

Una vez capturadas las imágenes, estas deben ser procesadas por un ordenador. En nuestro caso, hemos optado por hacer un procesamiento a través de Matlab, utilizando *POO* para agilizar el desarrollo de software y facilitar la creación de programas y prototipos visuales. Es aquí donde se concentra un gran porcentaje del trabajo real que ha tenido este Proyecto, ya que son necesarias cientos de líneas de código para conseguir detectar movimientos tan sutiles como el pestañeo de un ojo, el giro de la cabeza hacia un lado u otro, la apertura o cierre de la boca, etc.

Nuestro robot físico cuenta con servomotores que lo habilitan para movilizar seis facciones de la cara: ojo izquierdo, ojo derecho, ceja izquierda, ceja derecha, boca y giro de la cabeza. De esta manera, el tratamiento de imagen que hacemos en Matlab crea una nube tridimensional de 1347 puntos alrededor de la cara, que se actualizan a tiempo real rastreando cada movimiento que se realice (8).

Continuando de forma lineal el desarrollo del proyecto, una vez los datos son identificados a tiempo real por nuestro programa de Matlab han de ser enviados a la aplicación *Puppeteer*, desarrollada por la empresa del mismo nombre con la que hemos trabajado. Esta aplicación recibe directamente la información desde Matlab a través de OSC (*Open Sound Control*).

Siguiendo la que hubiera sido la línea original del Proyecto, el último paso hubiera sido continuar haciendo uso de OSC para enviar la información desde la aplicación de *Puppeteer* al robot real. De esta manera se completaría el ciclo: partiendo de la imagen captada por una cámara a tiempo real, y pasando por distintos interfaces de procesamiento y envío de información, el robot físico imitaría los movimientos realizados por el actor.

Sin embargo, al igual que ha ocurrido en prácticamente todos los ámbitos de la vida, el confinamiento causado por el Covid-19 también ha afectado al normal desarrollo de este trabajo. El robot físico y los dispositivos de conexión se encontraron en la Facultad en el momento de inicio de la cuarentena y, dado que no podemos acceder a ellos hasta comienzos del próximo curso, tuvimos que adaptarnos a la nueva situación para poder completar el proyecto. De esta manera es como entra en juego *Maya*, un programa informático dedicado al desarrollo de gráficos 3D por ordenador, donde se creó el modelo digital de nuestro robot. Así, el último paso del proyecto se ve un poco modificado: desde la aplicación de *Puppeteer* no se envía la información por OSC al robot físico, sino que se comunica con los canales de movimiento del robot en *Maya*, y la imitación de los movimientos faciales del actor se puede visualizar en la versión digital del robot, tal y como ocurre a la hora de dar vida a un personaje en una película de animación.

Si bien es cierto que en términos técnicos no encontramos una gran diferencia con lo que hubiera sido la línea original de trabajo, ya que el robot digital reproduce los movimientos de la misma manera que lo hubiera hecho el robot físico, en cuanto tengamos la oportunidad de acceder a él haremos las conexiones necesarias para poder observar en vivo el funcionamiento de todo el proceso.

1.3 Objetivos

Como ya hemos visto a lo largo de la Introducción, en este proyecto hemos tratado un tema denso, complejo y con bastante visión de futuro de cara a la industria cinematográfica. Con el presente proyecto se pretende crear la red de códigos y conexiones necesarias para realizar un procesamiento completo de imitación de expresiones faciales, partiendo desde el actor y llegando al robot. Con ello, se pretenden alcanzar los siguientes objetivos:

- En primer lugar, conocer el estado del arte actual sobre esta disciplina. Existen muchas personas y empresas trabajando para crear contenido animado innovador así que consideramos importante conocer con profundidad qué existe ya sobre este tema y qué está al alcance de un usuario particular antes de comenzar a crear nuestro propio código.
- Lograr acceder a propiedades muy específicas sobre las imágenes que se graben, ya que para su procesamiento necesitamos conocer información mucho más sofisticada de la que podamos obtener de una imagen normal.
- Crear un código en el lenguaje de programación de Matlab que sea capaz de reconocer, procesar e identificar rasgos y expresiones faciales con ligereza y fiabilidad.
- Establecer la red de conexiones necesarias para realizar la transmisión de información a tiempo real que necesitamos, haciendo uso de los protocolos o interfaces de conexión que resulten apropiados.
- Facilitar el programa obtenido a *Puppeteer Technologies*, empresa dedicada al mundo de la animatrónica 3D que ha colaborado con nosotros, y compartir con ellos cualquier tipo de resultado que obtengamos de cara a su implementación algún futuro proyecto cinematográfico.

- Conformar una base teórica y práctica sobre la Animatrónica en nuestra Escuela de Ingeniería. Como rama del Control y la Robótica, es una disciplina cuyo estudio aún no está implantado con totalidad en nuestra facultad, de forma que se esperan, por parte del departamento de Ingeniería de Sistemas y Automática, futuras líneas de trabajo en este ámbito. Los estudiantes de esta disciplina podrán tomar este proyecto como base, ya que se alojará en una página web como software de distribución libre, con la idea de promover el uso de la Animatrónica como plataforma de enseñanza en el Control y la Robótica.

2 RECURSOS EMPLEADOS Y CONEXIONES

En este apartado se va a describir, de forma técnica y teórica, el software y hardware ya existente que he necesitado para desarrollar mi proyecto, desde el dispositivo de grabación hasta el programa de modelado 3D, pasando por diferentes lenguajes, proyectos o aplicaciones que me han servido de base para alcanzar mi objetivo. Finalmente, voy a esquematizar las interconexiones que hay entre cada uno de estos elementos, para poder tener una idea clara de la estructura del proyecto.

2.1 Recursos empleados

2.1.1 Cámara Kinect v2 y 'Kinect SDK para Windows'

El punto de partida de todo el proyecto es la grabación de imágenes a tiempo real que contengan la información necesaria para realizar su correcto procesamiento. Para ello, tras un largo estudio sobre los diferentes tipos de cámaras a las que podíamos tener acceso, dimos con las cámaras *Kinect*, dispositivos asociados a las consolas *Xbox*. Estas cámaras se hicieron muy populares nada más salir al mercado, hace ya casi una década, ya que introdujeron una nueva característica de juego: el mando no era otro que el propio cuerpo del jugador. Las cámaras Kinect estaban dotadas de la capacidad de reconocer gestos, comandos de voz, objetos e imágenes, además de modelar nuestro propio avatar a nuestra imagen y semejanza. Fueron una vuelta de tuerca más en la industria del videojuego, revolucionando lo que se entendía hasta el momento como realidad virtual desde tu propia casa. Fue hasta tal punto la innovación de estos dispositivos que incluso ganaron premios como el *MacRober Award* o el *T3.gadget of the year award*, entre otros (9).

Es por esto que las aplicaciones de estas cámaras fuera del mundo de los videojuegos no tardaron en hacerse notar. Comenzaron a ser usadas para rastrear movimientos faciales y corporales de cara a proyectos de animación 3D o como controladores de efectos visuales y 'Projection mapping' en eventos relacionados con música electrónica y otras ramas del arte, entre otras muchas aplicaciones (10).

Existen dos modelos diferentes de cámaras Kinect, *Kinect v1* (asociada a la *Xbox 360*) y *Kinect v2* (asociada a la *Xbox One*). Ambas tienen características similares que las convierten en cámaras muy potentes para el procesamiento de imágenes. No obstante, además de eso cada una de ellas cuenta con su versión para Windows, desarrollada por Microsoft con vista a que se pudiera aprovechar su versatilidad para todo tipo de aplicaciones en el PC, desde interfaces para personas con problemas de movilidad hasta control de máquinas, sistemas de vigilancia o medicina (11).

En octubre de 2014, Microsoft unificó ambas versiones del *Kinect v2*: presentó un kit adaptador que permitía, por primera vez, anexar el *Kinect* para *Xbox One* al PC. Fue a partir de este momento cuando todos los sensores *Kinect v2* ('Kinect v2 para Windows' y 'Kinect para Xbox One') comenzaron a tener el mismo desempeño (12). Es más, el diseño de Kinect v2 para Windows es prácticamente idéntico al de la versión de Xbox One. Concretamente, las únicas diferencias se encuentran en el cambio de logo, las conexiones de entrada del núcleo (la versión original necesita un adaptador concreto para conectarlo al PC) y los cables de corriente.

A continuación, presentamos una tabla comparativa de las características que presentan ambos modelos de la cámara, *Kinect v1* y *Kinect v2*, en su uso para Windows.

Tabla 2-1. Tabla comparativa de las características de Kinect v1 y Kinect v2 para Windows (11)

Funciones	Kinect v1 (Xbox 360)	Kinect v2 (Xbox One)
Resolución	640x480, 30 fps, 4:3	1920x1080, 30 fps, 16:9
Ángulos de visión	57 grados horizontal, 43 grados vertical	70 grados horizontal, 60 grados vertical
Distancia mínima de uso	1.82 metros	1.37 metros
IR Activo (visión nocturna)	No	Sí
Latencia	102 ms	20 ms
Ajuste manual del motor	Sí	No
Detección simultánea de personas	4	6
Puntos del cuerpo simultáneos	20	25
Detección de dedos y muñecas	No	Sí
Detección de músculos	No	Sí
Medidor de pulsaciones	No	Sí

A día de hoy es prácticamente imposible hacerse con cualquiera de estas cámaras, dado que Microsoft dejó de fabricarlas hace varios años y ya no las comercializa. Aun así, para nuestro proyecto conseguimos hacernos con un dispositivo *Kinect v2* en la versión original para *Xbox One*. Elegimos la segunda versión tras analizar pormenorizadamente las características que la diferenciaban de la primera: mayor ángulo de visión, profundidad, resolución y, sobre todo, detección de un mayor número de puntos del cuerpo simultáneos, lo cual, en nuestro caso, mejoraría notablemente la calidad y precisión de la nube de puntos alrededor de la cara del actor.

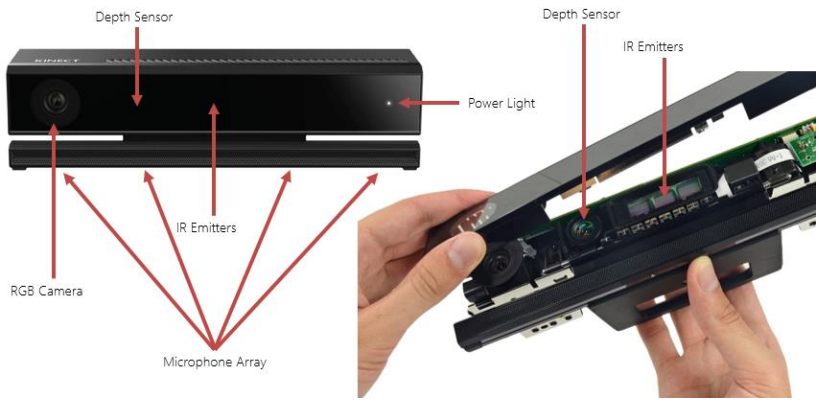


Figura 2-1. Localización de los distintos sensores en la Kinect v2 (9)

Al haber trabajado con el *Kinect v2* para *Xbox One* en el ordenador, tuvimos que conseguir, además, el adaptador para Windows. Este adaptador cuenta con dos hubs: uno para enchufar el cable USB 3.0 que manda los datos de *Kinect* al PC, y otro diferente para el cable de alimentación.

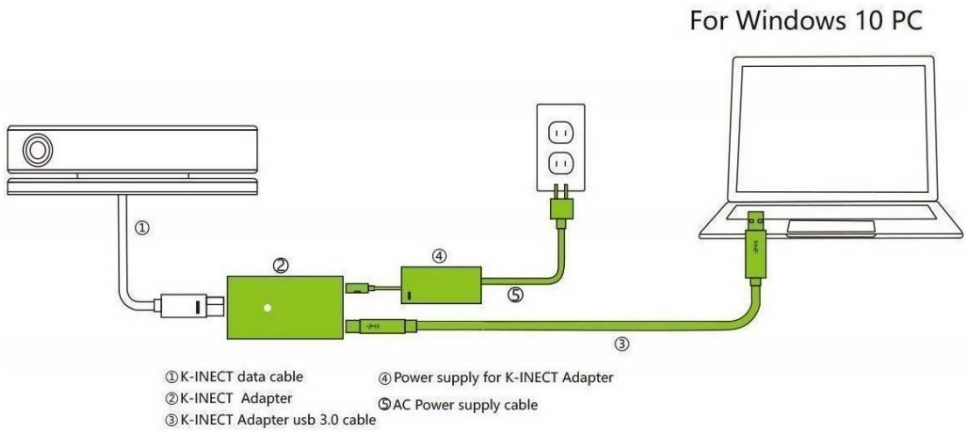


Figura 2-2. Esquema de cables y hubs del adaptador de Kinect v2 para Windows (13)

Una vez tenemos todo el hardware instalado, pasamos a estudiar el software que Microsoft ha desarrollado para *Kinect v2* para Windows.

Microsoft ha desarrollado una aplicación en Windows Store dedicada a Kinect: 'Kinect SDK 2.0'. Es un Kit de Desarrollo Software, una beta que fue lanzada por el área de investigación de la firma, que incluye todo lo que los desarrolladores necesitan para escribir, general, probar e implementar aplicaciones haciendo uso de las cualidades de Kinect v2. La descarga de *Kinect SDK 2.0* es gratuita y permitió, por primera vez en 2013, desplegar las aplicaciones de *Kinect* en Windows.

Este software de desarrollo incluye drivers, interfaces de programación de aplicaiones (APIs) para flujos de datos no procesados del sensor, natural user interfaces (NUI), archivos de instalación y materiales de referencia. El SDK ofrece a los desarrolladores la capacidad de diseñar aplicaciones *Kinect* con C++, C# o Visual Basic® a través de 'Microsoft Visual Studio 2010'.

Entre las características más destacables del SDK, podemos mencionar las siguientes (14):

- Facilidad de instalación. Se instala de forma rápida, no requiere configuración compleja y el tamaño total de instalación es de menos de 100MB. Además, incluye un verificador de configuración, que analiza el PC en busca de incompatibilidades de hardware y verifica la correcta comunicación con el sensor Kinect.
- Documentación extensiva. El SDK incluye más de 100 páginas de documentación técnica de alta calidad, que incluye explicaciones para la mayoría de muestras incluidas en el SDK y archivos de ayuda integrados.
- Flujos de datos no procesados del sensor. Los desarrolladores tienen acceso a flujos de datos no procesados de los cuatro micrófonos, del sensor de profundidad y del sensor de la cámara a color e infrarroja.
- Detección del esqueleto. El SDK tiene la capacidad de detectar el esqueleto de una o dos personas en movimiento dentro del campo de visión de Kinect, haciendo posible crear aplicaciones basadas en la gesticulación.
- Detección facial. Incluye demostraciones de la capacidad del sensor de detectar la cara del usuario, crear una nube de puntos en movimiento alrededor de ella y analizar su orientación.
- Capacidades avanzadas de audio. Las capacidades de procesamiento de audio incluyen supresión sofisticada de ruido y cancelación de eco, formaciones de haces para la identificación de la fuente actual de sonido e integración con el API de reconocimiento de voz en Windows.

Todas estas propiedades hacen del SDK para Windows una herramienta prometedora de cara a la implementación de las aplicaciones del *Kinect* en el ordenador.

2.1.2 Matlab

A la hora de codificar el procesamiento de imágenes, y teniendo en cuenta que este programa servirá de base a futuros estudiantes de animatrónica en la universidad, decidimos que sería importante crearlo en un lenguaje que se estudie con profundidad a lo largo de la carrera. De esta manera fue como decidimos codificar en Matlab en su última versión actualizada, 2020a, ya que además los universitarios disponemos de licencia de estudiantes para descargarlo y trabajar con total facilidad.

Con respecto al tipo de programación, nos decidimos por la Programación Orientada a Objetos frente a la Programación Estructurada, pese a ser esta última la realmente estudiada a lo largo del grado. La POO cuenta con una serie de ventajas con respecto a la Programación Estructurada que, de cara a la funcionalidad de nuestro código, la hacen mucho más favorable.

Algunas de las características que nos inclinaron hacia la POO son las siguientes (16):

- Reusabilidad. Una vez las clases están adecuadamente diseñadas, se pueden utilizar en distintas partes del programa y en numerosos proyectos, tal y como buscamos con este trabajo.
- Mantenibilidad. Debido a la sencillez de abstracción del problema, los programas orientados a objetos son más sencillos de leer y comprender, ya que nos permiten ocultar detalles de implementación dejando visibles sólo aquellos detalles más relevantes.
- Modificalidad. Presenta una gran facilidad para añadir, suprimir o modificar objetos.
- Fiabilidad. Al dividir el problema en partes más pequeñas, es posible probarlas de manera independiente y aislar de manera sencilla los posibles errores que pudieran surgir.

De esta manera, mi programa de visión artificial está estructurado, a grandes rasgos, en tres clases que se pueden esquematizar jerárquicamente de la siguiente manera:

Figura 2-3. Esquema jerárquico de las clases del programa de reconocimiento de imágenes

Como explicamos en el apartado anterior, el dispositivo *Kinect v2* tiene múltiples aplicaciones en lo que a reconocimiento de imágenes se refiere. Estas aplicaciones están disponibles para PC a través de su SDK. Sin embargo, por parte de Microsoft no existe el desarrollo de estas aplicaciones para el lenguaje de programación concreto que utiliza Matlab.

Haciendo una investigación sobre los trabajos ya publicados sobre este tema, di con una publicación que suplía por completo las necesidades que me surgían llegados a este punto. Diana M. Córdova-Esparza y Juan R. Terven, investigadores mexicanos, crearon en 2016 lo que titularon un 'Toolbox para Matlab de Kinect v2'. Este Toolbox encapsula la mayor parte de las funcionalidades del 'Kinect SDK para Windows' en una sola clase que contiene métodos de alto nivel de codificación, además de cualidades añadidas acerca de la calibración de la cámara y nubes de puntos de color. Su paper introduce este Toolbox de Matlab para la *Kinect v2*, llamado *kin2*, que facilita el desarrollo de aplicaciones con estas funcionalidades en Matlab sin necesidad de sumergirse en funciones en C# o C++ (8).

Kin2 está en su mayoría implementado en C++, de manera que se combina con Matlab a través de una función MEX. Un archivo MEX es una función, creada en Matlab, que llama a un programa de C/C++, comportándose como una función o un script de Matlab. En Windows x64, la extensión del archivo MEX es 'mexw64'. Sin embargo, para llamar a una función MEX sencillamente se utiliza el nombre del archivo sin la extensión; la sintaxis de llamada depende de los argumentos de entrada y salida que define la función MEX, debiendo esta estar en su ruta de Matlab (17).

La estructura del Toolbox *Kin2* está compuesta de cuatro niveles diferenciados, como podemos observar en la siguiente figura.

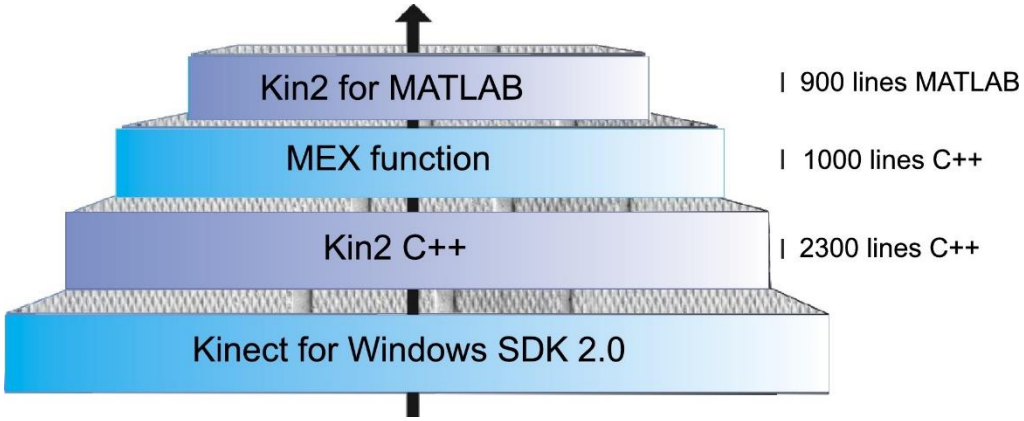


Figura 2-4. Diagrama piramidal de la arquitectura de kin2

En la base se encuentra lo que sería el corazón del Toolbox: el Kit de Desarrollo Software de Microsoft para el *Kinect v2*. Como desarrollamos con más detalle en el apartado anterior, este SDK provee al investigador de una serie de clases y funciones orientadas al desarrollo de aplicaciones para el *Kinect v2* en C# y C++, usando *Visual Studio 2012*. En el siguiente nivel de la jerarquía encontramos una clase, *Kin2*, escrita en 2300 líneas de código en C++. Esta clase encapsula las funcionalidades del SDK oficial, constituyendo así una interfaz entre las variables de C++ y las variables de Matlab. En el siguiente nivel encontramos ya la función MEX de Matlab, que no hace sino servir de interfaz entre la totalidad del código en C++ y el código en Matlab, dando pie así al nivel superior, la clase *kin2* para Matlab. Esta clase implementa todas las funcionalidades del *Kinect v2* en una clase de Matlab de tan sólo unas 900 líneas de código, consiguiendo así finalmente instaurar en Matlab las facilidades de visión que aporta este dispositivo.

El sensor Kinect v2 proporciona múltiple fuentes de información, que si recordamos era uno de los objetivos iniciales descritos del proyecto. La información sobre las imágenes que nos puede aportar puede ser color, profundidad, infrarrojos, marcos de detección de puntos corporales o rastreo facial; incluso podría llegar a utilizarse un mismo sensor para varios cuerpos al mismo tiempo. La función de inicialización conecta el sensor y establece un lector multi-fuente con las entradas definidas en los parámetros de entrada. Estas posibles entradas serán: 'color', 'depth','infrared','body' y 'face' o 'HDface'. Según las entradas que se envíen al llamar a la clase kin2, esta nos proporcionará una información u otra (8).

Las funcionalidades del *Kinect v2*, que ahora somos capaces de desarrollar desde Matlab, se pueden agrupar en cinco grandes grupos:

- Adquisición de imágenes. Los métodos de adquisición de imágenes proveen acceso al color, profundidad y cámaras infrarrojas a tiempo real; también proporcionan etiquetado corporal para seguimiento de las siluetas.
- Mapeo de coordenadas. Se basa en el mapeo de puntos en profundidad y su conexión con el mapeo de puntos en color, y el traslado de puntos entre la imagen 2D (coordenadas de cada punto) y la cámara en 3D.
- Rastreo corporal. Rastreo de hasta seis cuerpos al mismo tiempo, con localización de 25 articulaciones, análisis de la posición de las manos y representación corporal en color o profundidad.
- Procesamiento facial. Tiene dos niveles de profundidad. El procesamiento más sencillo analiza ocho propiedades faciales y rastrea cinco puntos de la cara del actor (ambos ojos, nariz y comisuras de la boca) a tiempo real. El nivel avanzado realiza una captura de movimientos faciales con 94 unidades de deformación facial, y rastrea un modelo facial que consta de 17 unidades animadas, en un máximo de seis personas dentro del plano del Kinect.
- Reconstrucción 3D. A través de lo que se conoce como 'Kinect Fusion', se realiza una reconstrucción 3D de escenarios estáticos.

Así, en la clase *kin2* de Matlab podemos encontrar una gran cantidad de métodos creados para abarcar todas estas funcionalidades. Sin embargo, no todas ellas forman parte del objetivo de nuestro proyecto. De hecho, la única que realmente nos interesa es la de procesamiento facial, en sus dos niveles, que nuestra cámara puede realizar. De esta manera, pese a que tenemos acceso a todos los métodos de la clase *kin2*, en realidad sólo hacemos uso de seis de ellos como base para poder desarrollar todo nuestro programa. Vamos a profundizar un poco más en ellos.

1. updateData

Se llama a esta función siempre antes de grabar datos nuevos. Actualiza los datos de la Kinect y nos devuelve una bandera indicadora de si los datos son válidos, permitiéndonos en este caso avanzar en el código.

```
function varargout = updateData(this, varargin,

varargout{1:nargout}]=Kin2_mex('updateData',this.objectHandle,varargin{:});

end
```

2. getColor

Una vez asegurado que los datos recibidos son válidos, se pasa a su procesamiento. Esta función, tras comprobar que la información se seleccionó correctamente, nos devuelve tres marcos de color de tamaño 1920x1080. Al igual que la anterior, también se ejecuta con cada imagen captada.

```
function varargout = getColor(this, varargin)
   % Verificar que la información se seleccionó correctamente
   if ~this.flag_color
        this.delete;
        error('No color source selected!');
   end

   varargout{1:nargout}] =
Kin2_mex('getColor',this.objectHandle,varargin{:});

end
```

3. getFaces

Constituye el primer nivel del procesamiento facial. Esta función es originalmente capaz detectar y rastrear la cara, e identificar ocho propiedades faciales: feliz, preocupado, con gafas, ojo izquierdo cerrado, ojo derecho cerrado, boca abierta, boca en movimiento y mirada perdida. Este método ejecuta código en C++, a través de la función MEX, para extraer la información del *Kinect* y la plasma en una estructura, que es lo que devuelve la función.

La estructura que nos devuelve contiene la siguiente información para cada una de las caras detectadas:

Tabla 2-2. Contenido que nos devuelve la función la función *getFaces*

Nombre	**Descripción**
FaceBox	Coordenadas de los cuatro vértices del rectángulo que encuadra la cara detectada.
FacePoints	Vector [2x5] que contiene las coordenadas *(x,y)* de cinco puntos específicos de la cara del actor: centro ojo izquierdo, centro ojo derecho, centro nariz y comisuras de la boca.
FaceRotation	Vector [1x3] que contiene la orientación de la cara, expresada en los tres ángulos de Euler.
FaceProperties	Vector [1x8] con los valores de las ocho propiedades anteriormente descritas; cada una puede tener los siguientes valores: Desconocido (0), No (1), Quizás (2), Sí (3)

```
function varargout = getFaces(this, varargin)

   % Verificar que la información se seleccionó correctamente
     if ~this.flag_face
         this.delete;
         error('No face source selected!');
     end

[varargout{1:nargout}] = Kin2_mex('getFaces', this.objectHandle,
varargin{:});

end
```

4. getHDFaces

Esta función constituye el segundo nivel de procesamiento facial. Se caracteriza por presentar tres grandes cualidades:

- Captura facial: construye un modelo facial personalizado compuesto por 94 unidades de deformación facial, basándose en un modelo facial generalizado.
- Rastreo facial: se realiza al actor un rastreo facial de 17 unidades animadas (AUs) expresadas numéricamente a tiempo real con cifras entre [0,1].
- Modelo facial: se construye un modelo facial tridimensional y actualizado a tiempo real, formado por 1347 vértices.

Estos análisis se realizan al ejecutar la función *getHDFaces*. Como resultado, la función original nos devuelve una estructura con la siguiente información para cada una de las caras reconocidas:

Tabla 2-3. Contenido que nos devuelve la función *getHDFaces*

Nombre	Descripción
FaceBox	Coordenadas de los cuatro vértices del rectángulo que encuadra la cara detectada.
FaceRotation	Vector [1x3] que contiene la orientación de la cara, expresada en los tres ángulos de Euler.
HeadPivot	Vector [1x3] que devuelve el centro de la cabeza, alrededor del que la cara gira. El origen está localizado en el centro óptico del Kinect, el eje Z crece en dirección al usuario, el eje Y señala hacia arriba y el eje X señala hacia la derecha.
AnimationUnits	17 AUs expresadas con valores numéricos entre 0 y 1. Definen la forma de la cara y están definidas como propiedades al inicio de la clase kin2. Acceder a (15) para una lista completa de las AUs.
FaceModel	Modelo facial de alta definición con 1347 vértices mallados. Acceder a (18) para una lista de los puntos faciales en HD.

```
function varargout = getHDFaces(this, varargin)

   % Verificar que la información se seleccionó correctamente
   if ~this.flag_hd_face
       this.delete;
       error('No HDface source selected!');
   end

   [...]

[varargout{1:nargout}]=Kin2_mex('getHDFaces',this.objectHandle,withVertices);

end
```

5. drawFaces

Una vez está realizado todo el procesamiento de imagen para un instante concreto, es necesario reproducir en pantalla los resultados obtenidos. Es importante tener en cuenta que, al ser un procesamiento de imágenes a tiempo real, la muestra de resultados debe ser reproducida también a tiempo real, por lo que esta función no puede ser lenta ni difícil de digerir por parte de Matlab.

La función original *drawFaces* no devuelve nada, sencillamente trata de etiquetar cada uno de los valores analizados por *getFaces* en la pantalla de visualización en la que el usuario se está viendo a tiempo real. Es decir, para cada una de las caras reconocidas, se crea un cuadrado alrededor de ella (*FaceBox*), se observan cinco puntos rojos en los lugares de la cara anteriormente mencionados (*FacePoints*) y salen en la imagen los valores numéricos de los tres ángulos de Euler (*FaceRotation*) y la información sobre las ocho propiedades estudiadas (*FaceProperties*).

Para ello, es evidente que la función debe recibir toda esta información, y lo hace a través de una estructura llamada 'faces', que es exactamente la misma que nos devolvió *getFaces* anteriormente. Además de esta estructura, la función debe recibir el propio objeto (no olvidemos que estamos trabajando en POO) e información sobre el texto y la fuente de escritura en la imagen.

```
function drawFaces(this,handle,faces,pointsSize,displayText,fontSize)
```

No obstante, la función original de drawFaces que encontramos en la clase kin2 ha sido modificada para adaptarse a este proyecto. Originalmente esta función no tiene ningún argumento de salida. Sin embargo, como veremos con detalle más adelante, en nuestro caso necesitamos que nos devolviera determinada información, de manera que la función original fue modificada. El código implementado podemos encontrarlo en el Anexo 1.

6. drawHDFaces

Es una función análoga a la anterior, excepto que esta vez los valores que se representan en pantalla son los que devuelve la función *getHDFaces* (*FaceBox, FaceRotation, HeadPivot, AUs* y *FaceModel*). Recibe una estructura 'faces' con dicha información, características del texto y fuente de escritura y, además, los vértices del modelo facial tridimensional que crea la función.

```
function
drawHDFaces(this,handle,faces,displayPoints,pointsSize,displayText,fontSize)
```

Al igual que en su función análoga, drawHDFaces ha sido modificada para poder obtener de ella determinada información que necesitábamos con vistas a completar la red de conexiones para llegar al robot. La función finalmente implementada podemos encontrarla en el Anexo 2.

Como hemos podido comprobar, las cuatro primeras funciones de las que hacemos uso apenas tienen complejidad ninguna, ya que se limitan a llamar a la función MEX de Matlab de la que hemos hablado con anterioridad, que sirve de interfaz entre Matlab y el código original en C++. Con respecto a las dos funciones restantes, como veremos en el apartado 3. Clases en POO, no las hemos implementado en su forma original, sino que hemos modificado su contenido para adaptarlas a la finalidad de nuestro proyecto.

2.1.3 Puppeteer Studio

Puppeteer Studio es una aplicación de escritorio desarrollada por la empresa de animatrónica *Puppeteer Technologies*, que es la desarrolladora del middleware[1] que estamos usando para controlar el robot. Este sistema se dedica a facilitar la conexión entre sistemas físicos y dobles virtuales a software. Consiste principalmente en un entorno que nos permite configurar cada eje de movimiento, establecer límites y reglas de movimiento para poder mover el sistema con seguridad y nos proporciona unidad física de control que se conecta vía USB.

Esta aplicación permite, además, la generación y edición de plataformas, un seguimiento telemétrico a tiempo real del robot o la gestión y asignación de personajes animados, entre otras aplicaciones más. El software actualmente está en versión Alpha, y así lo hemos descargado y utilizado en este proyecto.

Puppeteer Studio está diseñada para soportar la conexión con un número ilimitado de canales, de manera que se puede realizar la comunicación entre los distintos miembros de un equipo utilizando estándares abiertos. La conexión se realiza a través de la IP, permitiendo así enviar y recibir información a tiempo real bajo el protocolo OSC, del que hablaremos un poco más adelante. Así mismo, está diseñada para comunicarse con Autodesk Maya/3ds Max, etcétera. La aplicación, en resumen, nos permite desarrollar de manera visual e intuitiva la producción completa de un personaje animado (6).

Pese a la infinidad de recursos que ofrece, si tenemos que enfocarnos en nuestro caso concreto, Puppeteer es el encargado de servir de interfaz para establecer la comunicación por OSC entre Matlab y nuestro robot.

[1] Middleware es un software que asiste a una aplicación para comunicarse con otras aplicaciones, paquetes de programas, redes, etc.

Primeramente, configuramos la aplicación para que se comporte como receptor OSC, especificando la IP y el puerto, como vemos en la siguiente imagen:

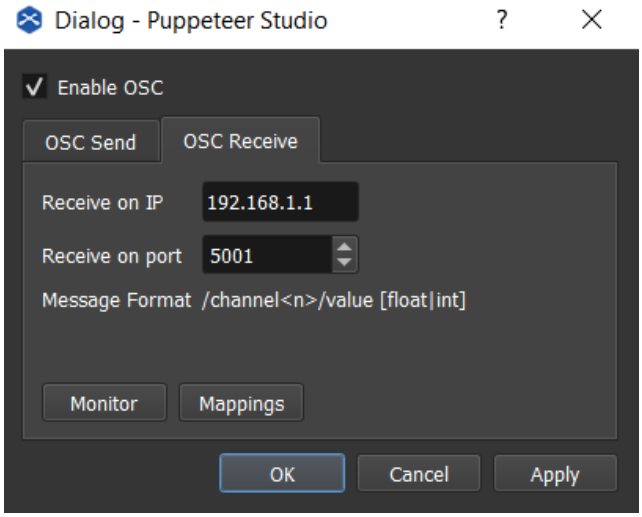


Figura 2-5. Ventana de conexión con OSC en Puppeteer

Una vez creada la conexión, los canales de Puppeteer aumentan o disminuyen su valor numérico según la información que nuestro programa de Matlab le envíe en cada instante. El rango numérico que cubren se encuentra entre [-1,1]. La cabeza de nuestro robot tiene ocho puntos de movimiento, que en Puppeteer se traducen como ocho canales diferentes de información.

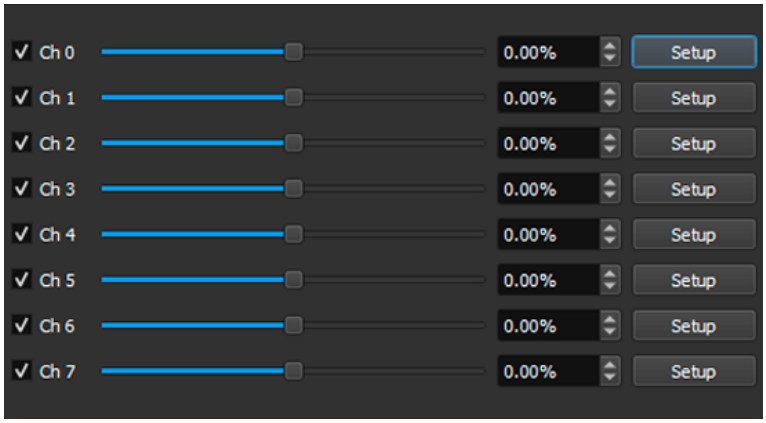


Figura 2-6. Ventana con canales de envío/recepción de datos vía OSC en Puppeteer

Así se completa la funcionalidad que esta aplicación tiene, por el momento, en nuestro proyecto: permitirnos crear una ventana de comunicación OSC para hacer llegar nuestra información al robot a tiempo real.

2.1.4 OSC

Los protocolos de comunicación son estándares técnicos que establecen especificaciones sobre métodos y procesos para implementar el intercambio de mensajes entre aplicaciones, hardware o ambas. Es decir, definen el formato de los mensajes que deben intercambiarse, cómo interpretarlos y cómo identificar al emisor y receptores de la comunicación. Pese a que se trata, en realidad, de un convenio de mensajes, muchos de estos protocolos están basados en la comunicación por red IP, y otros por transacciones específicas de paquetes de bytes directamente. Entre los protocolos más utilizados destacan MIDI, Art-Net, SNMP u OSC, en mayor parte usados en sistemas audiovisuales.

Podemos definir *Open Sound Control* como un protocolo abierto que define un formato de mensajes basado en facilitar la comunicación entre dispositivos capacitados para recibir o enviar datos por medio de la red, como pueden ser ordenadores, sintetizadores de sonido u otros controladores multimedia. OSC constituye un sistema de comunicación flexible y preciso a la hora de interoperar entre distintos dispositivos, permitiendo de esa manera el control a tiempo real de sonido y otros tipos de datos (19).

- Escenario genérico

A grandes rasgos, OSC se basa en un formato de mensajes que nos permite encapsular la información con valores determinados como argumentos. El formato de los mensajes OSC es tan abierto que nos permite idearlos y formularlos a nuestro gusto. Sin embargo, para poder recibir e interpretar estos mensajes es necesario generar una aplicación que sirva como servidor OSC, que en nuestro caso es la función que cubre la aplicación Puppeteer. Este servidor, como anunciamos anteriormente, es capaz de escuchar mensajes OSC a través de su dirección IP en la red y un número de puerto, encargándose también de gestionar la comunicación entre las distintas conexiones entrantes.

El servidor OSC también puede enviar mensajes, generando lo que se conoce como un cliente OSC y conectándose igualmente al IP y al puerto correspondiente. En caso de querer que la comunicación sea bidireccional entre dos o más aplicaciones, y así permitir que puedan tanto recibir como enviar mensajes, estas deberán contener un servidor y un cliente trabajando a la vez.

Para más información acerca de OSC, la fuente (20) concentra todo tipo de información específica sobre OSC, sus recursos, implementaciones y documentación específica.

- Características generales

Entre las características más relevantes de OSC, podemos hablar de las siguientes (19):

- Ampliable, dinámico. Estructura de mensajes simbólica tipo URL
- Transferencia de datos numéricos de alta resolución
- Utiliza lenguaje de coincidencia de patrones (‘Pattern matching’) para especificar múltiples receptores de un único mensaje
- Marcas de tiempo de alta resolución
- Mensajes ‘empaquetados’ para generar eventos simultáneamente
- Sistema de consulta para encontrar dinámicamente las capacidades de un servidor OSC y su documentación

- Áreas de aplicación

Tanto OSC como el resto de protocolos de comunicación de este tipo están principalmente orientados a la comunicación entre instrumentos electrónicos y el mapeo de datos entre aplicaciones o instrumentos musicales. Sin embargo, al ser un protocolo basado en comunicación por la red, el abanico de aplicaciones es mucho más amplio que esto. Otra gran aplicación consiste en servir de sistema de control centralizado y compartido entre diferentes actores a tiempo real sobre una interfaz compartida, además de servir de interfaz web: establecer comunicación y transferir datos desde y hacia una aplicación web. También se utiliza como conversor de formatos de datos, ya que es capaz de convertir un protocolo de datos específico a OSC, dada su facilidad de transmisión por la red y la flexibilidad de formulación de mensajes.

Es por todas estas cualidades y aplicaciones por lo que la posibilidad que nos ofrece *Puppeteer* de entablar una comunicación vía OSC con nuestro robot, mejora enormemente la calidad, facilidad y rapidez de ejecución de nuestro proyecto, permitiendo observar a tiempo real los resultados en el robot.

2.1.5 Autodesk Maya

Autodesk Maya es un programa informático dedicado al desarrollo de gráficos 3D por ordenador, efectos especiales, animación y dibujo computarizado. Maya surgió como evolución de *Power Animator* y de la fusión de *Alias* y *Wavefront*, dos empresas canadienses dedicadas a los gráficos generados por ordenador. Finalmente *Autodesk* las absorbió, siendo a día de hoy el desarrollador de *Maya*.

Maya conforma una herramienta muy potente para el desarrollo tridimensional computarizado, caracterizada principalmente por su potencia y por las posibilidades de expansión y personalización de su interfaz. El código que forma el núcleo de Maya se denomina MEL (*Maya Embedded Language*), y es gracias a él que se pueden crear scripts y personalizar paquetes (21).

Como ya adelantamos anteriormente, el papel de *Maya* en nuestro proyecto surgió de manera forzada tras la situación creada por la pandemia. Originalmente, la aplicación *Puppeteer* se iba a poner en contacto directamente con el robot físico. Sin embargo, al no haber tenido acceso al robot por el desarrollo de este proyecto durante la cuarentena, tuvimos que hacer uso de su doble digital en *Maya*. Así, *Puppeteer* envía información a nuestro robot en *Maya* vía OSC, como se explicó en el apartado anterior, y este imita los movimientos del actor exactamente igual que lo hará el robot físico cuando podamos tener acceso a él, siendo este el procedimiento real que siguen los personajes de animación para cine.

La recepción de los paquetes de datos en Maya se visualiza en ocho canales diferentes, cada uno correspondiente con los ocho canales de envío de información que tiene Puppeteer.

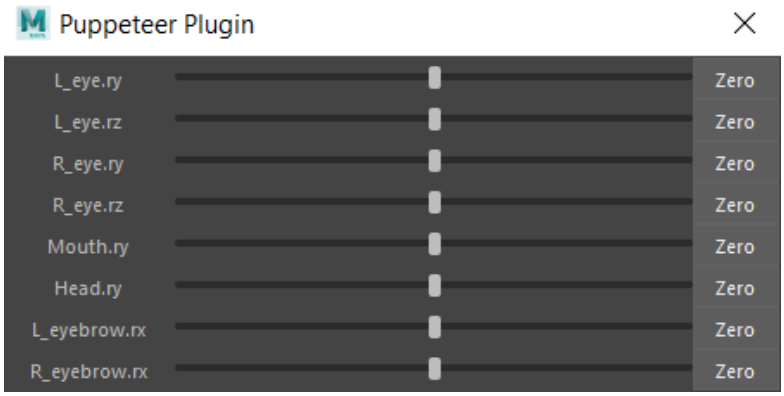
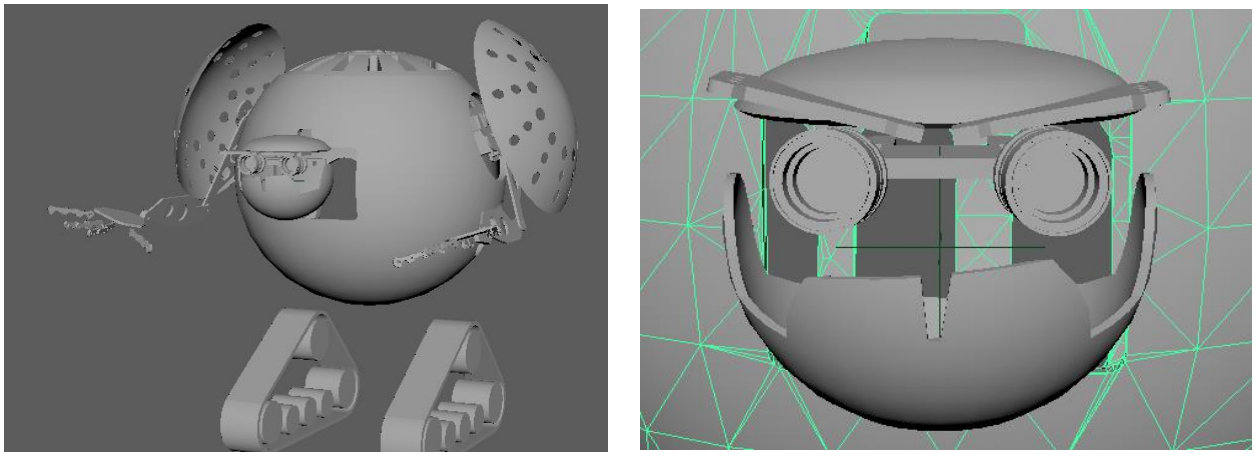


Figura 2-7. Representación de los canales de movimiento en la aplicación de Maya

En la figura 2-7, todos los canales se encuentran en posición neutra. Es decir, con valor cero. Recibirán información numérica contenida entre [-1,1], y según el valor que tengan, la facción del robot a la que corresponde cada canal tendrá una posición u otra. Esto se verá detalladamente más adelante.

La versión de nuestro robot en Maya tiene el aspecto que podemos observar en las siguientes figuras.



a) Visualizado del cuerpo entero del robot b) Visualizado de las facciones de la cara

Figura 2-8. Versión digital de nuestro robot en Autodesk Maya

2.2 Conexiones entre dispositivos

Una tarea determinante en nuestro Proyecto es la correcta conexión entre cada una de los dispositivos hardware o software que trabajan durante el envío de la información: desde el movimiento del actor hasta el movimiento del robot.

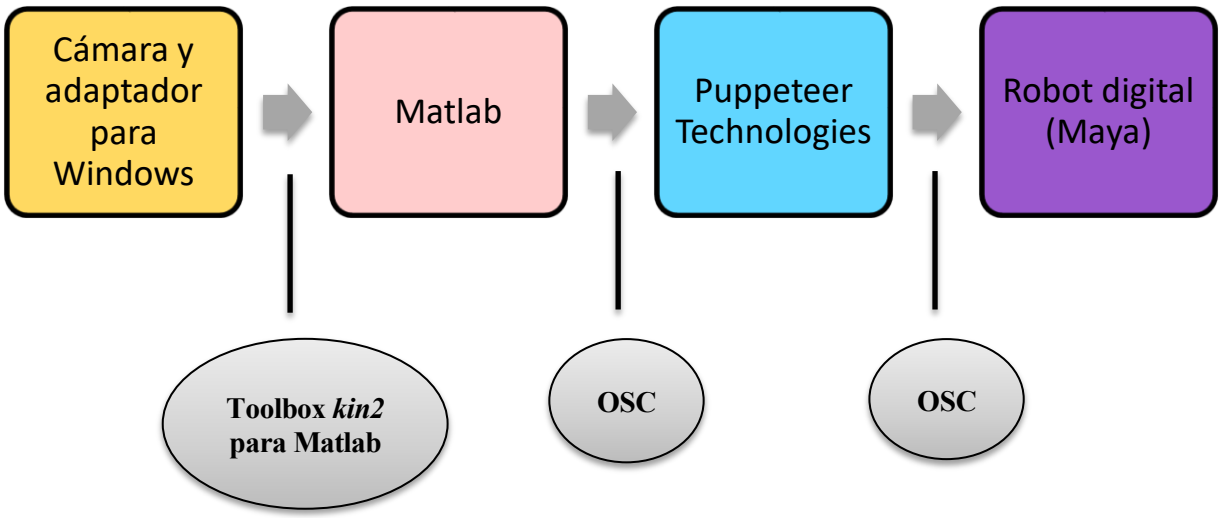


Figura 2-9. Red de conexiones del proyecto

Este es el esquema general de nuestro proyecto, que iremos desarrollando conforme vayamos definiendo nuevas características del mismo.

- Tras instalar correctamente la cámara y el SDK en el PC con el que se vaya a trabajar, la conexión con Matlab la realizamos gracias al Toolbox *kin2*, que nos permite hacer uso de las facilidades de la cámara en Matlab.
- Dentro de *Matlab* se desarrollará el grueso del proyecto, que se añadirá al esquema más adelante. La salida de datos de *Matlab* hacia la aplicación *Puppeteer Technologies* se produce a través de OSC, siendo esta la segunda conexión.
- La última conexión necesaria para completar el proceso es la salida de paquetes de datos desde *Puppeteer* hacia *Maya*. Esta conexión también se realiza vía OSC.

3 CLASES EN POO

Una vez llegados a este punto, ya hemos presentado pormenorizadamente qué elementos, aplicaciones y programas hemos utilizado en nuestro proyecto y tenemos una idea esquematizada de cuál es el papel de cada uno de ellos dentro del mismo. Por lo tanto, a lo largo de este apartado se van a presentar las clases creadas en Matlab y la interconexión que existe entre ellas, que es lo que conforma el núcleo central de todo el proyecto.

Para ello, primeramente vamos a volver a presentar el esquema general de nuestro proyecto, pero esta vez uniendo toda la información que tenemos al respecto, para crear una idea de los bloques con los que contamos dentro del Matlab.

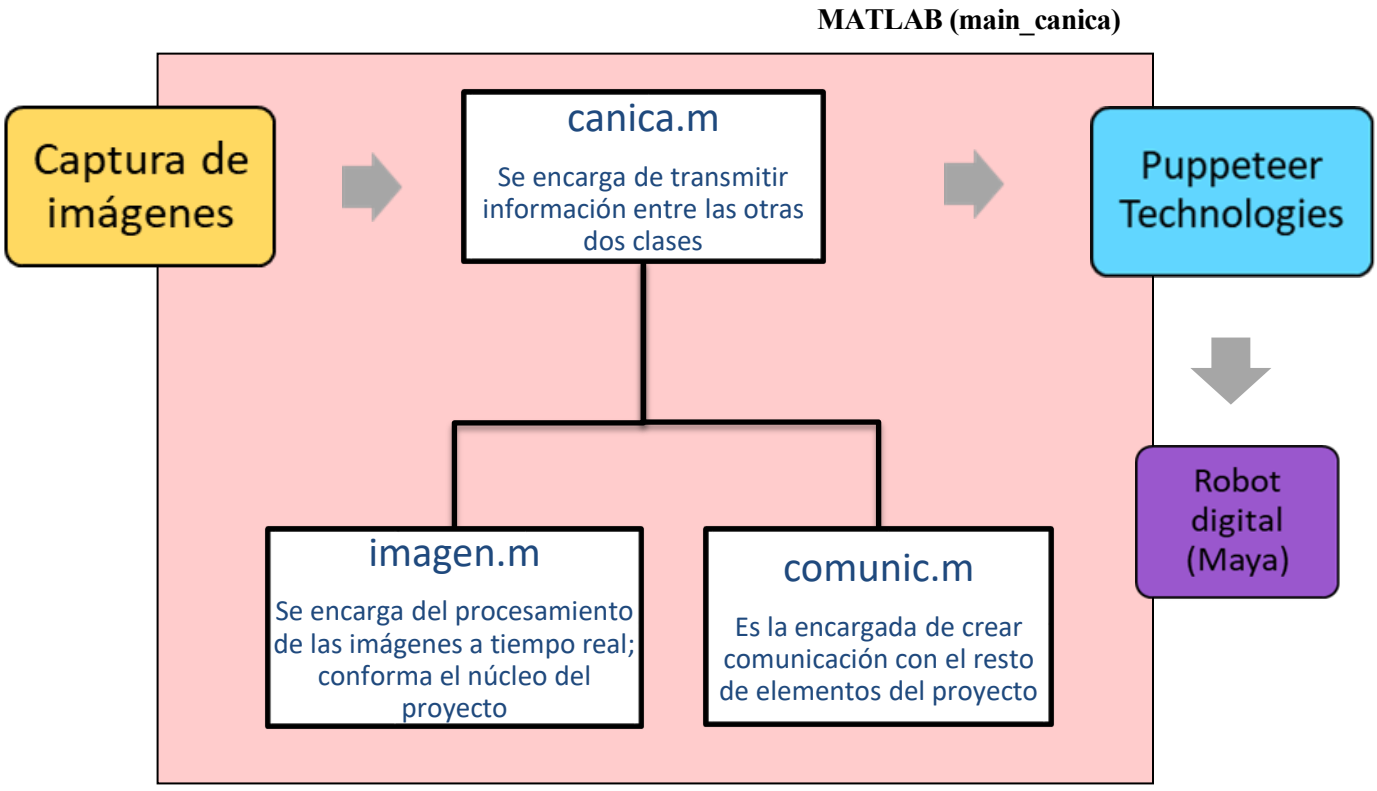


Figura 3-1. Esquema ampliado de la red de conexiones del proyecto

Entendemos, así, que el TFG se puede dividir en dos grandes bloques:

- Imágenes: incluye los métodos y funciones encargados de la grabación, tratamiento y procesamiento de imágenes a tiempo real.
- Comunicación: conforma el código y los programas externos a Matlab que crean la red de transmisión de información, desde el actor hasta el robot.

Podemos decir que la clase principal de nuestro código es la clase *canica*. Esta clase contiene la formación general del código que trata la grabación de imágenes y, además, es la principal transmisora de información dentro de Matlab, por lo que en ella se encuentran los núcleos de los dos bloques arriba descritos.

Al estar trabajando mediante POO, todas las clases necesitan ser instanciadas en algún momento, para tener un objeto que las contenga. Dentro de esta clase principal, *canica*, se crean los objetos de las otras dos. Sin embargo, precisamos de una función básica que instancie el primer objeto, el de la clase *canica*, que será lo que de pie al inicio de todo el proceso.

3.1 Función main_canica

```
addpath('Mex');

clear all
close all
clc

%1.Crear objeto e inicializar

a=canica ();

%2. Producir la grabación (imagen.m -> canica.m -> comunic.m -> ROBOT)

choice = menu('Presiona el botón para comenzar la identificación de
imagen','Comenzar');

if choice==1

   grabar(a);

end
```

La función *main_canica* es esa función original de la que hablamos. Como podemos observar, lo primero que hace esta función es agregar las carpetas especificadas (en nuestro caso, la que contiene los archivos .mex) a la parte superior de la ruta de búsqueda en Matlab.

Una vez ubicados, crea el objeto que inicializa todo el proceso. Al crear el objeto, "a", se llama a la 'Constructor Function' de la clase *canica*, que se encarga de inicializar todo el proceso (lo veremos con detalle a continuación).

Una vez ya creado el objeto, comienza la grabación y procesamiento de imágenes llamando a la función *grabar* de *canica*. Antes de esto se despliega un menú que espera hasta que el usuario responda, de manera que evitamos que comience un proceso de grabación y tratamiento de imágenes si el actor aún no está delante de la cámara.

3.2 Clase canica

Como ya hemos adelantado, es la clase principal que coordina a las otras dos. A grandes rasgos, contiene las líneas de código que procesan los dos grandes bloques de nuestro trabajo: imágenes y comunicación, como veremos detalladamente a continuación.

No obstante, hay que tener claro desde el primer momento de qué manera se realiza un procesamiento de imágenes a tiempo real en un programa como Matlab. Como sabemos, un vídeo no es más que una consecución de imágenes a una frecuencia muy alta. Un tratamiento de imágenes a tiempo real viene a ser lo mismo: cada 'frame' (imagen) que la cara captura, se analiza. Si la cámara tiene una frecuencia de 30fps, según la tabla 2-1, teóricamente se analizaría la información de 30 'capturas' por segundo. Ya veremos más adelante que, en la práctica, la frecuencia de capturas por segundo es algo más baja.

El primer paso en toda clase es crear las propiedades, que serán las que darán forma a la clase. Es importante aclarar aquí que, al igual que las clases *comunic* e *imagen* se crean como propiedades de *canica*, la clase *kin2* de la que hablamos anteriormente también se crea como una de las propiedades de *canica*. Sin embargo, recordamos que *kin2* necesita entradas diferentes según vayamos a trabajar con *Faces* o con *HDFaces*. Como en nuestro caso vamos a trabajar con las dos, sencillamente creamos dos propiedades diferentes, una llamada *k2* y otra llamada *k2HD*, y a cada una de ellas se le asignan las fuentes de entrada apropiadas.

Una vez creadas las propiedades, comenzamos a desarrollar el contenido de la clase a través de los métodos y funciones. En el caso de nuestra clase principal, el primer método contiene la función constructora, que se encarga de llamar, a su vez, a las funciones constructoras de las otras dos clases para crear sus objetos y así inicializar el proceso.

```
methods

    function obj = canica()

        obj.comunic = comunic(); %Llama a la Constructor Function de comunic
        obj.imagen = imagen();% Llama a la Constructor Function de imagen

    end
end
```

Nuestra clase *canica* tan sólo contiene una función más, que es a la que se llama desde *main_canica* como *grabar*, y que tiene una relevancia extrema en la totalidad del proyecto, por lo que vamos a explicar con detalle sus principales líneas de código a continuación.

La primera parte de la función se encarga de dar valores numéricos a las propiedades anteriormente creadas y de formar una figura, *figure (1),* en la que más adelante saldrá la imagen grabada por la cámara con el título de 'Face Recognition', aunque que ahora mismo se limita a sacar una imagen en negro. Además, esta primera parte pone a cero tres de las propiedades, *i, j* y *flag*, que nos servirán de contadores para iniciar y finalizar el procesamiento de imágenes.

La segunda parte del método *grabar* se desarrolla dentro de un bucle *while true*. Recordamos que, hasta este momento, ya se ha creado una figura llamada 'Face Recognition' que está mostrando una imagen en negro. Así mismo, recordamos las seis funciones de la clase *kin2* que describimos con detalle en apartados anteriores porque iban a ser usadas en nuestro código: *updateData, getColor, getFaces, gatHDFaces, drawFaces, drawHDFaces.*

Ahora sí, una vez dentro de nuestro bucle llamamos a la función ***updateData*** y comprobamos si la información que nos devuelve es válida. En caso de que la información no sea válida, sencillamente se vuelve al inicio del bucle *while* y se vuelve a solicitar la información hasta que sea procesable. En el momento en que los datos recibidos tanto para *faces* como para *HDfaces* sean válidos, entramos dentro de un bucle *if*. Se actualiza la información de la imagen, llamando a la función ***getColor*** de *kin2*, y ahora sí lo que vemos en la figura 'Face Recognition' comienza a ser la imagen que la cámara está captando; es decir, la cara del actor.

```
while true

    % Se llama a updateData
    obj.validData = obj.k2.updateData;
    obj.validDataHD = obj.k2HD.updateData;

    % Se comprueba que los datos recibidos sean válidos
    if obj.validData && obj.validDataHD

        % Se actualiza la figura con la imagen del actor
        obj.color = obj.k2.getColor;

        obj.color = imresize(obj.color,obj.COL_SCALE);

        figure(1),imshow(obj.color, 'Parent', obj.c_ax);
        title('Face Recognition');
```

A continuación se realiza el análisis de la cara llamando a las funciones ***getFaces*** y ***getHDFaces*** de la clase *kin2* y guardando dicha información en dos de las propiedades que se crearon al inicio, *obj.faces* y *obj.HDfaces*. Una vez guardada esta información, llamamos a las únicas dos funciones de kin2 que, según estudiamos anteriormente, nos faltan por utilizar: ***drawFaces*** y ***drawHDfaces***, que imprimen en la figura 'Face Recognition' toda la información que se ha analizado en las líneas inmediatamente superiores.

```
% Obtener información de la cara

   obj.faces = obj.k2.getFaces;
   obj.HDfaces=obj.k2HD.getHDFaces('WithVertices','true');

% Sacar por pantalla la información obtenida

   obj.k2.drawFaces(obj.c_ax,obj.faces,5,true,20);
   obj.k2.drawHDFaces(obj.c_ax,obj.HDfaces,true,true,20);
```

Hasta ahora el bucle *while* ha tenido objetivos muy claros: actualizar los datos y, si son válidos, sacar por pantalla la cara del actor, analizarla y añadir a la imagen los resultados analizados.

La última parte del código de este bucle ha ido siendo modificada conforme se ha ido probando y actualizando el código. Encontrábamos el error de que se analizaba e imprimía por pantalla (getFaces y drawFaces) la información de absolutamente todos los frames que la cámara obtenía. En muchos casos, hasta que el código era capaz de reconocer la cara del actor por primera vez, la información que se imprimía por pantalla era totalmente indiferente: todos los valores salían con valor desconocido, ya que al programa aún no le había dado tiempo de identificar la cara. Es por esto que se desarrollaron varios bucles que aseguraran que, una vez comenzara el envío de información entre clases, fuera porque ya se había detectado la cara del actor, de manera que no se iba a ejecutar código en vano. Es aquí donde entran en juego las propiedades *flag, i, j* que inicializamos anteriormente a cero.

Una vez la cara del actor se ha reconocido por primera vez, *flag*=1.

```
if obj.flag==0 & obj.size_faces==[1 1] & obj.size_HDfaces==[1 1] %Detecta
la primera vez que se reconoce la cara

   obj.flag=1;

end
```

Dentro del bucle *while* encontramos tres líneas de código que presentan un nuevo bucle *if*: aumentan en 1 el valor de j cada vez que se ejecuta el bucle *while*, pero sólo en caso de que *flag*=1 (es decir, ya se haya reconocido la cara por primera vez) y además en esta vuelta del bucle también se haya reconocido la cara (es poco frecuente, pero a veces ocurre que aunque ya se haya captado la cara, esta se pierde, por lo que ese frame tampoco sería un frame válido para enviar a analizar).

```
if obj.flag==1 & obj.size_faces==[1 1] & obj.size_HDfaces==[1 1] %Aumenta j
con cada frame que reconoce la cara, una vez flag=1

    obj.j=obj.j+1;

end
```

El objetivo de este bucle *if* es muy claro: no queremos enviar la información de cada uno de los frames que la cámara recoja, ya que hemos entendido que el actor necesita al menos unos segundos para cambiar su expresión facial. De esta forma, el frame que se 'retendrá' y se enviará a analizar será aquel en el que j=10: se habrán detectado 10 frames válidos, durante los cuales el actor habrá podido asegurarse de tener la expresión facial deseada. Al décimo frame válido, (j=10), se retiene la imagen y se envía al resto de clases para ser traducida a valores entendibles por el robot y, finalmente, imitada por el mismo.

```
if obj.j>=10 & obj.size_faces==[1 1] & obj.size_HDfaces==[1 1]

   obj.i=obj.i+1;
   obj.j=0; %El contador de frames válidos se pone a 0, para volver a
             comenzar la cuenta


   obj.values=obj.imagen.analisis(obj.k2,obj.c_ax,obj.faces,obj.HDfaces);
   obj.comunic.sender(obj.values,1000);

end
```

Aunque pueda parecer que el tiempo transcurrido hasta el primer envío de información al robot es muy largo, realmente es un instante lo que tarda en ocurrir; lo suficiente para que el actor se asegure de que se está captando su expresión correctamente antes de ser enviada. Una vez esto ocurre, *j=0* y el contador *i* aumenta en 1. Este contador nos dará el número total de frames que van a ser enviados al robot. Es decir, cuántas expresiones queremos que se imiten.

En el rodaje de una película de animación este último índice no sería fructífero; en su lugar habría que usar, quizás, un contador temporal del tiempo que dure la escena, o incluso dejarlo abierto hasta que manualmente se pare el proceso. Dado que este código ha sido experimental, el hecho de poner un número de 'escenas' a imitar por el robot nos permitía estudiar su eficacia con mucha más facilidad.

De esta forma, el bucle *while* acaba con un último condicionante *if*, que es donde está contenido el 'break' que nos sacará del código: cuando ya se hayan imitado 10 expresiones faciales (i=10), pongo los marcadores a 0 y salgo del bucle *while*, finalizando así el código.

```
if obj.i==10 %N° total de frames que le voy a enviar al robot

   obj.flag=0;
   obj.i=0;
   break;

end
```

Esta es la explicación detallada del funcionamiento de la función *grabar* de la clase *canica*. Sin embargo, hay dos líneas de código cruciales sobre las que no hemos hecho ningún comentario. Al principio dijimos que la clase *canica* conformaba el núcleo de nuestro código, ya que trabajaba para los dos bloques de nuestro proyecto: tratamiento de imágenes y comunicación entre clases y aplicaciones. Todo el código que hemos analizado de esta clase hasta ahora ha estado orientado a la grabación, análisis y procesamiento de imágenes, pero sin embargo no hemos explicado ninguna línea dedicada a la comunicación entre clases.

Volviendo atrás, recordamos que estas eran las líneas en las que se captaba un frame para enviarlo e imitarlo por parte del robot. Tras actualizar los valores de los contadores, encontramos las dos líneas de código siguientes:

```
obj.values=obj.imagen.analisis(obj.k2,obj.c_ax,obj.faces,obj.HDfaces);

obj.comunic.sender(obj.values,1000);
```

Son las encargadas de realizar la comunicación entre clases, posicionando así a la clase *canica* en el nivel superior en la jerarquía del diagrama de la figura 3-1: es la encargada de transmitir la información entre las tres clases. Primeramente, se llama a la función *análisis* de la clase *imagen*, que veremos más adelante: se le envía la información que tenemos de nuestro procesamiento (*obj.faces* y *obj.HDfaces)*. Esta función realiza la 'traducción' entre lo que nosotros le enviamos y valores numéricos que nuestro robot es capaz de entender y reproducir. Como resultado de esta traducción, nos devuelve un vector *values* que contiene dichos valores.

Una vez tenemos este vector, se lo enviamos a la función *sender* de la clase *comunic*, que también explicaremos más adelante. Es en este punto donde terminan las responsabilidades de la clase *canica*: una vez se asegura de que *comunic* tiene esta información, es ella la encargada de hacérsela llegar al robot a través de protocolos y aplicaciones que ya no son controlados por *canica*.

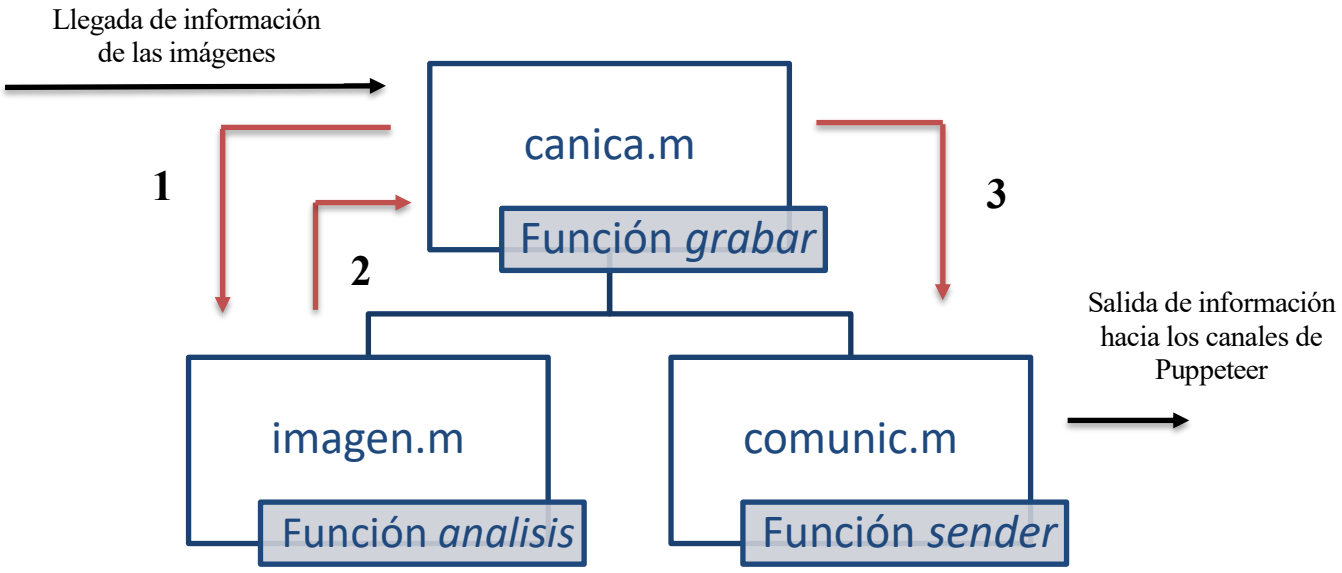


Figura 3-2. Diagrama del flujo de información dentro del programa de Matlab

3.3 Clase imagen

Como ya sabemos, el objetivo de esta clase es 'traducir' la información que le envía *canica* sobre el estado facial en cada frame, a valores numéricos que sean comprensibles por nuestro robot.

Al igual que en cualquier clase, antes de nada debemos crear las propiedades que darán forma a nuestra clase, que en este caso son solamente tres, pero jugarán un papel importante en el desarrollo de las funciones.

```
properties

   facciones; %vector [1x8] que devolveremos
   infofaces,infoHDfaces;

end
```

Una vez hecho esto, pasamos a la función constructora. En el caso de esta clase, la función constructora tiene una único objetivo muy simple: se encarga de inicializar a cero el vector *facciones*, que es una de las propiedades creadas anteriormente en esta clase, y que más tarde será donde estará contenida la información numérica de las facciones de la cara analizadas.

```
methods

function obj_imagen=imagen()

   obj_imagen.facciones=zeros(1,8);

end

end
```

Una vez creadas las propiedades y la función constructora pasamos a crear la función *análisis*, que si recordamos fue la función a la que llamamos anteriormente desde la función *grabar* de la clase *canica* y nos devolvió un vector de 8 caracteres con valores numéricos que pasamos directamente como argumento a la clase *comunic*.

El contenido de esta nueva función está claramente dividido en dos partes: obtención de información (llamando a *drawFaces* y *drawHDFaces*) y traducción de esta información a valores numéricos legibles por el robot. El código de la primera parte es de la siguiente manera:

```
methods

  function valores = analisis(obj_imagen,k2_im,c_ax_im,faces_im,HDfaces_im)

     valores=zeros(1,8); % Inicialización del vector a 0

     %Faces
      obj_imagen.infofaces=zeros(1,5);

      obj_imagen.infofaces=k2_im.drawFaces(c_ax_im,faces_im,5,true,20);

      obj_imagen.facciones(1)=obj_imagen.infofaces(1);%OJO IZQU. UP-DOWN
      obj_imagen.facciones(2)=obj_imagen.infofaces(2);%OJO IZQU. CERRADO?
      obj_imagen.facciones(3)=obj_imagen.infofaces(3); %OJO DRCHO. UP-DOWN
      obj_imagen.facciones(4)=obj_imagen.infofaces(4); %OJO DRCHO. CERRADO?
      obj_imagen.facciones(5)=obj_imagen.infofaces(5);%BOCA ABIERTA/CERRADA
```

```
%HDFaces

  obj_imagen.infoHDfaces=zeros(1,3);

  obj_imagen.infoHDfaces=k2_im.drawHDFaces(c_ax_im,HDfaces_im,true,
  true,20);

  obj_imagen.facciones(6)=obj_imagen.infoHDfaces(1); %HEAD YAW
  obj_imagen.facciones(7)=obj_imagen.infoHDfaces(2); %CEJA IZQU.
  obj_imagen.facciones(8)=obj_imagen.infoHDfaces(3); %CEJA DRCHA.
```

Lo primero que se hace es inicializar a cero el vector [1x8], para asegurarnos de que no nos dará error al ocuparlo llamando a otras funciones. A continuación, vamos a proceder a obtener la información que tiene que tener cada uno de los ocho elementos del vector. Esta información la obtenemos como resultados de las clases ***drawFaces*** y ***drawHDFaces***, que son funciones de la clase *kin2* que ya fueron llamadas con anterioridad desde la clase *canica* para imprimir por pantalla a tiempo real información sobre las expresiones faciales del actor. En este caso, la información que nos devuelven no se va a observar en ningún lugar, sino que simplemente va a ocupar las ocho posiciones del vector facciones.

De la función ***drawFaces*** obtenemos cinco valores que nos interesan, así que los almacenamos en el vector [1x5] llamado *infofaces*, que fue creado como propiedad de esta clase con este cometido. Una vez este vector está completo, se traspasa la información a los cinco primeros elementos del vector *facciones*. Igualmente se procede con la función ***drawHDFaces*****:** nos devuelve tres valores, que primeramente almacenamos en *infoHDFaces* y de ahí los pasamos a las tres últimas posiciones de nuestro vector *facciones*.

La información que almacena cada uno de estos ocho elementos del vector, y cuál es la función de *kin2* que nos facilita a cada uno de ellos, podemos verlas con claridad en la siguiente figura:

Tabla 3-1. Información sobre los ocho elementos del vector facciones

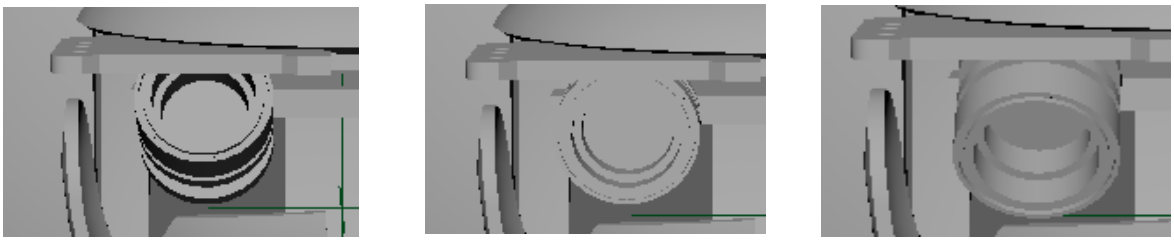
Elemento del vector *facciones*	**Información que contiene**	**Función de *kin2* que nos da la información**
1	Nivel de altura de la pupila del ojo izquierdo	*drawFaces*
2	Ojo izquierdo abierto/cerrado	*drawFaces*
3	Nivel de altura de la pupila del ojo derecho	*drawFaces*
4	Ojo derecho abierto/cerrado	*drawFaces*
5	Boca abierta/cerrada	*drawFaces*
6	Ángulo de giro de la cabeza	*drawHDFaces*
7	Nivel de elevación de la ceja derecha	*drawHDFaces*
8	Nivel de elevación de la ceja izquierda	*drawHDFaces*

La primera parte de las dos que hablamos antes ya está cumplida: hemos obtenido toda la información que necesitábamos. Es el momento de comenzar con la segunda parte de la función *análisis*, que consiste en traducir los ocho valores numéricos que tenemos a valores que nuestro robot vaya a poder entender. Para ello es necesario aclarar que estos valores tendrán que estar contenidos entre [-1,1] de forma que estén dentro del rango de movimiento del robot y así se produzca correctamente la imitación de los movimientos. Esta 'traducción' está totalmente personalizada a cada una de las seis facciones que se analizan, por lo que el código es diferente en cada una de ellas. Es importante saber, antes de pasar a explicar cómo se realizan estas traducciones numéricas, que el famoso vector que la función devuelve se llama *valores* y tiene ocho elementos que, como hemos dicho, serán valores numéricos contenidos entre [-1,1].

3.3.1 Ojo izquierdo

Recordamos que los elementos (1) y (2) del vector *facciones* nos daban información sobre el ojo izquierdo: nivel de la pupila en el eje vertical, y ojo cerrado o no (0: desconocido, 1: abierto, 2: quizás esté cerrado, 3: cerrado).

Además, el elemento del vector *valores* que contendrá el valor entre [-1,1] con la información del ojo izquierdo será el primer elemento *(valores (1))*. El posible contenido numérico que le demos a este elemento puede ser -1, 0 ó 1, según estudiemos que el ojo izquierdo esté cerrado, mirando al frente o mirando hacia arriba:



a) Ojo hacia arriba b) Ojo al frente c) Ojo hacia abajo

Figura 3-3. Tres posiciones posibles del ojo izquierdo del robot

De esta forma, el estudio que se ha hecho de la pupila es el siguiente:

```
if obj_imagen.facciones(1)>85 && obj_imagen.facciones(2)~=3 &&
obj_imagen.facciones(2)~=0

   valores(1)=1;
   disp('Ojo izqu mirando hacia arriba');

elseif obj_imagen.facciones(1)<=85 && obj_imagen.facciones(1)>=78 &&
obj_imagen.facciones(2)~=3 && obj_imagen.facciones(2)~=0

    valores(1)=0;
    disp('Ojo izqu mirando al frente');

elseif obj_imagen.facciones(2)==3 %Ojo izqu. cerrado

    valores(1)=-1;
    disp ('Ojo izqu cerrado');

elseif obj_imagen.facciones(2)==0 %Ojo izqu. desconocido

    valores(1)=0;
    disp ('No hay información exacta del ojo izqu');

end
```

Se trata de localizar la pupila dentro de la cara haciendo cuatro preguntas diferentes. Hay dos preguntas que son condicionantes del resto: si el ojo está cerrado (*facciones (2) = 3*), directamente sacamos por pantalla ese valor y el robot lo imitará (*valores (1) = -1*). Si no tenemos información exacta del ojo (*facciones (2) = 0*), imprimiremos esta información por pantalla y el ojo del robot no ejecutará ningún movimiento. En caso de que la pupila esté suficientemente alta (*valores (1) > 85*) Y además nos aseguremos de que el ojo no está cerrado, determinaremos que el ojo izquierdo está mirando hacia arriba (*valores (1) = 1*). Esto es así porque alguna vez, pese a reconocer el ojo cerrado, ha dado valores de pupila altos, por lo que eliminamos este posible error añadiendo la condición de ojo cerrado al if. Finalmente, si reconoce a la pupila entre unos límites experimentalmente normales ([78,85]), y de nuevo el ojo no está cerrado, concluimos que el ojo izquierdo está mirando al frente, y enviamos el valor apropiado al robot para que lo imite (*valores (1) = 0*).

3.3.2 Ojo derecho

El procedimiento de análisis del ojo derecho es análogo al realizado para el ojo izquierdo, con la diferencia de que en este caso trabajamos con *facciones (3)* y *facciones (4),* y que el valor devuelto se almacenará en *valores (2).*

Los tres posibles valores que le vamos a enviar al robot serán tres: ojo cerrado (*valores (2) = -1*), ojo mirando al frente (*valores (2) = 0*) y ojo mirando hacia arriba (*valores (2) =1*).

a) Ojo hacia arriba b) Ojo al frente c) Ojo hacia abajo

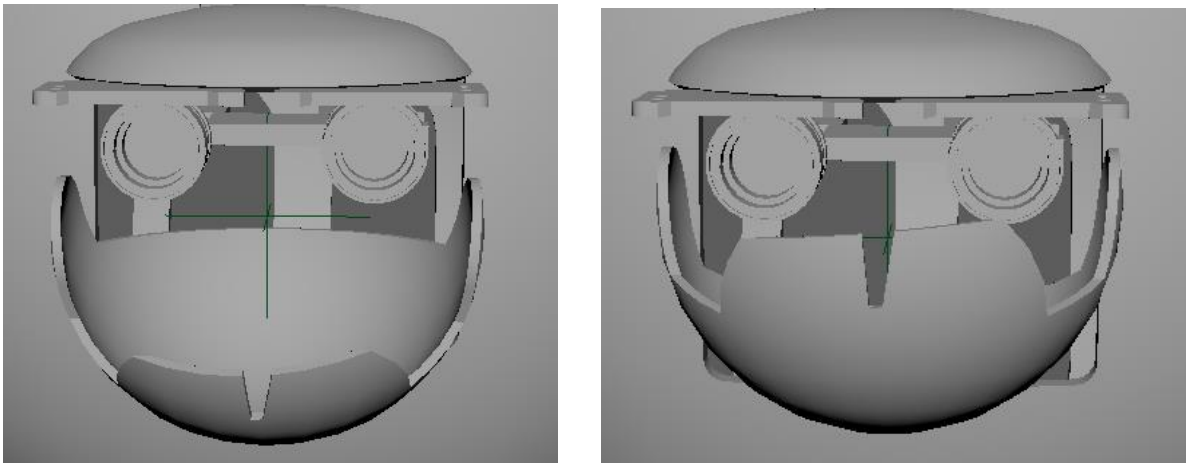
Figura 3-4. Tres posiciones posibles del ojo derecho del robot

El código que lo analiza es igual que el anterior, cambiando únicamente los elementos de cada vector:

```
if obj_imagen.facciones(3)>85 && obj_imagen.facciones(4)~=3 &&
obj_imagen.facciones(4)~=0

     valores(3)=1;
     disp('Ojo drcho mirando hacia arriba');

elseif obj_imagen.facciones(3)<=85 && obj_imagen.facciones(3)>=75 &&
obj_imagen.facciones(4)~=3 && obj_imagen.facciones(4)~=0

     valores(3)=0;
     disp('Ojo drcho mirando al frente');

elseif obj_imagen.facciones(4)==3 %Ojo drcho. cerrado
     valores(3)=-1;
     disp ('Ojo drcho cerrado');

elseif obj_imagen.facciones(4)==0 %Ojo drcho. desconocido
     valores(3)=0;
     disp ('No hay información exacta del ojo drcho');

end
```

3.3.3 Boca

En el caso de la boca, el análisis se simplifica bastante, ya que como vimos en la tabla 3-1, el valor de *facciones (3)* nos dice directamente si la boca está abierta, cerrada, o si no hay información exacta sobre la boca. En el caso de que se reconozca que la boca está abierta, haremos *valores (5) = -1*. Si la boca está cerrada, o si no hay información exacta de la misma, *valores (5) = 0*, que dejará la boca del robot en posición cerrada.



a) Posición boca abierta b) Posición boca cerrada

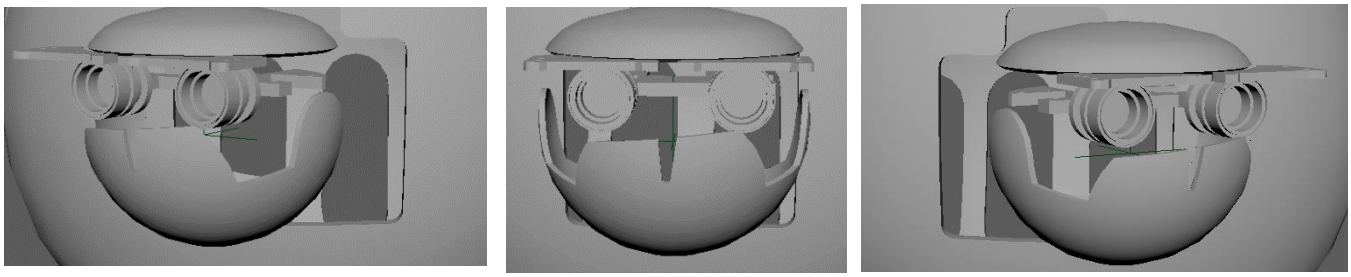
Figura 3-5. Posiciones de la boca

Siendo el código que lo analiza de la siguiente forma:

```
if obj_imagen.facciones(5)==1 %Boca cerrada

     valores(5)=0;
     disp('Boca cerrada')

elseif obj_imagen.facciones(5)==3 %Boca abierta

     valores(5)=-1;
     disp('Boca abierta')

elseif obj_imagen.facciones(5)==0 || obj_imagen.facciones(5)==2 %No hay
información exacta

     valores(5)=0;
     disp('No hay información exacta de la boca')

end
```

3.3.4 Cabeza

El estudio que se realiza en este caso es el ángulo de giro de la cabeza hacia izquierda o derecha, expresado como uno de los ángulos de Euler. Para recibir esta información tenemos que fijarnos en el valor numérico de *facciones (6),* y lo traduciremos de nuevo en el valor -1, 0 ó 1 para el elemento *valores (6).* De esta forma, las posibles posiciones de la cabeza serán las siguientes:



a) Cabeza girada hacia la izqu. b) Cabeza al frente c) Cabeza girada hacia la derecha

Figura 3-6. Posiciones de giro de la cabeza

Y el código que realiza la 'traducción' el siguiente:

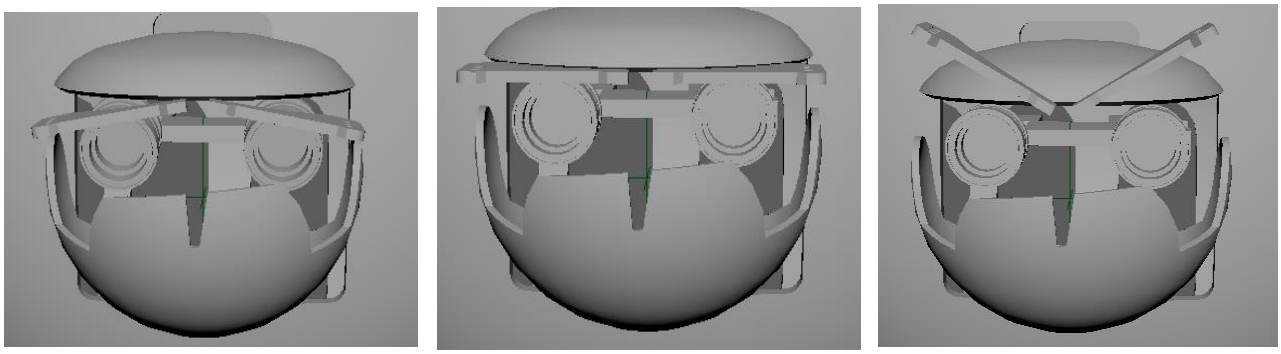
```
if obj_imagen.facciones(6) > -20 && obj_imagen.facciones(6) < 0 ||
obj_imagen.facciones(6) > 0 && obj_imagen.facciones(6) < 20

     valores(6)=0;
     disp ('Cabeza en el centro')

elseif obj_imagen.facciones(6) <= -20 %Cabeza hacia la derecha

     valores(6)=1;
     disp('Cabeza hacia la derecha')

elseif obj_imagen.facciones(6) >= 20 %Cabeza hacia la izquierda

      valores(6)=-1;
      disp('Cabeza hacia la izquierda')

end
```

Experimentalmente nos dimos cuenta de que para que la cabeza del actor esté centrada, el ángulo de salida debe estar comprendido entre [-20,20]. En caso de que el ángulo sea mayor o menos que estos valores, significará que la cabeza está claramente girada hacia uno de los dos lados.

3.3.5 Ceja izquierda

Las cejas son las dos últimas facciones de la cara del actor que se van a estudiar. En el caso de la ceja izquierda, la información la encontramos en *facciones (7),* y esta vez se trata de números entre [-1,1] pero con valores muy pequeños y ajustados incluso a la centésima, por lo que igualmente tenemos que hacer pruebas experimentales para estudiar los límites que realicen una correcta 'traducción' de estos valores a valores adecuados para el robot.

Las tres posiciones de las cejas, según enviemos la cifra que se envíe a *valores (7),* se ven de la siguiente manera:



a) Cejas hacia abajo b) Cejas horizontales c) Cejas hacia arriba

Figura 3-7. Posiciones de las cejas

Como vemos, las cejas son un elemento facial que aporta muchísima expresividad, por lo que su procesamiento es importante, pese a que no es especialmente sencillo ni exacto. Hay que aclarar que, en el caso de la ceja izquierda baja, el valor en lugar de ser -1, es -0.5, ya que sino la bajada de la ceja es de casi 90º.

El código que procesa la ceja izquierda es el siguiente:

```
if obj_imagen.facciones(7) > 0.2

    valores(7)=-0.5;
    disp ('Ceja izqu baja')

elseif obj_imagen.facciones(7) <= 0.2 && obj_imagen.facciones(7) >= -0.1

    valores(7)=0;
    disp ('Ceja izqu normal')

elseif obj_imagen.facciones(7) < -0.1

    valores(7)=+1;
    disp ('Ceja izqu alta')

end
```

Es decir, de manera aproximada conseguimos delimitar que si el valor recibido de altura de la ceja es mayor que 0.2, la ceja estará hacia abajo (*valores (7) = -0.5),* si está entre [-0.1,0.2], la ceja estará en una posición horizontal (*valores (7) = 0*), y en caso de que el número obtenido sea menor que -0.1, la ceja estará levantada (*valores (7) = 1*).

3.3.6 Ceja derecha

El análisis de la ceja derecha es el mismo que el arriba explicado, con la diferencia de que el valor que tenemos que analizar en este caso está guardado en *facciones (8),* y el resultado que obtengamos habrá que meterlo en *valores (8).* Al igual que ocurría con la ceja izquierda, en caso de que se detecte que la ceja derecha está bajada, el valor que se envía es de 0.5 en lugar de 1, para que la bajada de la ceja sea más o menos mesurada (ambas cejas tienen los valores positivos y negativos invertidos; la ceja izquierda baja en el robot con valores negativos, mientras que la ceja derecha baja con valores positivos).

Las tres posiciones de las cejas son las que vimos en la figura 3-6. El código que la analiza sigue el mismo método que el de la ceja izquierda pero, como hemos dicho, los análisis de las cejas son muy específicos, por lo que incluso cambian los valores de los límites entre una ceja y otra, tal y como podemos ver a continuación:

```
if obj_imagen.facciones(8) > 0.5

   valores(8)=0.5;
   disp ('Ceja drcha baja')

elseif obj_imagen.facciones(8) <= 0.5 && obj_imagen.facciones(8) >= 0.1

   valores(8)=0;
   disp ('Ceja drcha normal')

elseif obj_imagen.facciones(8) < 0.1

   valores(8)=-1;
   disp ('Ceja drcha alta')

end
```

3.4 Clase comunic

Recordamos que a la clase comunic la llamamos desde la clase principal, canica, y que le enviamos el vector con valores numéricos, ya traducidos, que nos ha devuelto la función análisis de la clase imagen:

```
obj.values=obj.imagen.analisis(obj.k2,obj.c_ax,obj.faces,obj.HDfaces);

obj.comunic.sender(obj.values,1000);
```

Al igual que las clases anteriores, el primer paso es definir las propiedades. En este caso tenemos ocho propiedades, que son las ocho posibilidades de movimiento que tiene nuestro robot, y que se definen de la siguiente forma:

```
properties

    l_eye_ry,l_eye_rz {mustBeNumeric};
    r_eye_ry,r_eye_rz {mustBeNumeric};
    l_eyebrow,r_eyebrow {mustBeNumeric};
    mouth {mustBeNumeric};
    head {mustBeNumeric};

end
```

Una vez definidas las propiedades, se define la función constructora de la clase. Como ya comentamos anteriormente, esta función constructora inicializa el robot a cero. Es decir, envía al robot un vector de ocho ceros, para que el robot adquiera la expresión más neutra posible. Todos los envíos de información al robot se hacen a través de una función llamada oscsend, que podemos ver en el Anexo 3, y que se encarga de crear la conexión OSC para hacer llegar la información a la aplicación Puppeteer Technologies, y de ahí al doble digital del robot en Maya.

```
methods %Constructor Function

     function obj_comunic = comunic()

      %Facciones a 0 en mis propiedades
       inic=[0,0,0,0,0,0,0,0];
       valorfaccion(obj_comunic,inic);
```

Es importante que el valor numérico que tengan las facciones en el robot en Maya en cada momento coincida con el valor que tienen las propiedades de las facciones de nuestra clase. Esto puede resultar un poco complicado de ver, pero al estar trabajando con POO, podría ocurrir que enviáramos por OSC los valores al robot, pero que las propiedades no se actualizaran. Para asegurarnos de esto, hemos creado una función relacionada con la clase llamada valorfaccion. A esta función se le envía el objeto y el valor numérico de las 8 facciones, y se encarga de actualizarlas:

```
function valorfaccion(obj,valores)

    obj.l_eye_ry=valores(1);
    obj.l_eye_rz=valores(2);
    obj.r_eye_ry=valores(3);
    obj.r_eye_rz=valores(4);
    obj.mouth=valores(5);
    obj.head=valores(6);
    obj.l_eyebrow=valores(7);
    obj.r_eyebrow=valores(8);

end
```

Así, lo primero que hace la Constructor Function de comunic es actualizar el valor de las propiedades a cero. Hay que darse cuenta de que esto actualiza las propiedades dentro de la clase, pero no actualiza los movimientos de la cara del robot para ponerlos en posición neutra. Para ello hay que crear el protocolo osc, que es lo que se hace en las líneas siguientes.

```
    %Facciones a 0 en el robot
    u = udp('127.0.0.1',5001); %IP y puerto
    fopen(u);

    for servo=0:7

        s=nombreservo(servo);
        oscsend(u,s,'i', 0);
    end

    fclose(u);

    end
end
```

Primero, se crea un objeto udp de Matlab. Como vamos a utilizarlo para comunicarnos con una aplicación externa (en nuestro caso, Puppeteer Technologies), debemos especificar el 'remote host' y el número de puerto del instrumento que va a recibir nuestra información. En nuestro caso, el receptor tiene la IP 127.0.0.1 y el número de puerto 5001. Una vez creado este objeto udp, hay que 'abrirlo' para poder comenzar la transmisión de paquetes de datos, que sí que será lo que acabe movilizando las facciones del robot.

Recordamos que estamos en la Constructor Function, cuyo objetivo era inicializar todas las facciones. Por ello, los valores que tenemos que enviar por osc son cero para los ocho servos que tenemos. Hacemos esto de manera rápida con un bucle for, que también hace uso de una función relacionada con la clase a la que hemos llamado nombreservo. A esta función le llega el número del servo que queremos actualizar, y devuelve el código que le corresponde a ese servo en Puppeteer, para que pueda reconocerlo:

```
function s=nombreservo(n) %Me relaciona el valor numérico del servo al que
quiero acceder con el nombre (texto) que tengo que enviarle a Puppeteer

            if n==0
            s='/servo0';
            elseif n==1
            s='/servo1';
            elseif n==2
            s='/servo2';
            elseif n==3
            s='/servo3';
            elseif n==4
            s='/servo4';
            elseif n==5
            s='/servo5';
            elseif n==6
            s='/servo6';
            else
            s='/servo7';
            end

end
```

Una vez enviados los valores de los ocho servos a 0, podremos observar que el robot cambia de expresión facial y pone todas las facciones en un punto neutro. La función última que realiza este envío de información se llama oscsend (22). A esta función le llega como argumento de entrada el objeto u, que como vimos anteriormente especifica la IP y el puerto de recepción de información, el canal de Puppeteer con el que nos estamos comunicando (tiene que tener el nombre que se haya definido desde Puppeteer), el tipo de dato que le vamos a enviar (entero, decimal, …) y, finalmente, el valor numérico deseado. Esta función crea la conexión OSC y hace que el canal especificado de Puppeteer cambie de valor.

Una vez inicializadas tanto las propiedades (mediante la función valorfaccion) como los canales de Puppeteer (mediante las funciones nombreservo y oscsend), la Constructor Function ha cumplido con su objetivo.

La función principal de esta clase es la función sender, de la que ya hicimos uso desde canica anteriormente. Esta función tiene exactamente la misma forma que la función constructora, con la única diferencia de que esta vez los valores que se envían no serán todos cero, sino que habrá que leerlos del vector que nos devolvió imagen y que le llegó a comunic como argumento de entrada.

```
methods %Ordinary Method: Member Functions

     function sender(obj_comunic,values,loop)
            u = udp('127.0.0.1',5001);

            fopen(u);
```

```
    for k=1:loop

        for servo=0:7

        s=nombreservo(servo);
        oscsend(u,s,'f', values(servo+1));

        end

    end

    fclose(u);

    %Actualizar el valor que acaba de cambiar de la variable del objeto
    valorfaccion(obj_comunic,values);

end
```

En cada envío de información por OSC se actualizan los valores de las 8 facciones, aunque no todas ellas hayan cambiado desde el envío anterior. Los dos únicos detalles añadidos con respecto al código de la función constructora son los siguientes. Primeramente, como argumento de entrada le llega un valor de 'loop'. Esto es así porque el robot realiza la imitación de los movimientos a una velocidad tan alta que apenas somos capaces de observar la trayectoria de cambio de posición. Al meterle el loop, sencillamente el movimiento se produce más lentamente y somos capaces de visualizarlo, pero no influye en ningún otro punto del proceso. La otra diferencia que observamos es que los valores de las propiedades se actualizan al final del proceso, también mediante la función *valorfaccion*, y esta vez entregándole como argumento de entrada el vector *values* en lugar de enviar todo ceros como se hizo en la función constructora.

Podemos ahora entender por qué definimos la clase *comunic* como aquella clase encargada de crear nexos entre Matlab y el resto de aplicaciones que forman parte de nuestro proyecto.

4 RESULTADOS Y ANÁLISIS

Por la forma del proyecto, los resultados en formato imagen son claros y fáciles de presentar y de analizar. Debemos ejecutar el programa y comprobar si, efectivamente, se cumplen nuestros objetivos: que el robot sea capaz de imitar a tiempo real los movimientos faciales del actor. No obstante, también debido al tipo de proyecto, la presentación de los resultados en este formato se convierte en una tarea algo más complicada, ya que una parte importante del proyecto consiste en conseguir ejecutarlo a tiempo real. Es por esto que a continuación vamos a presentar en formato fotográfico distintos reconocimientos e imitaciones faciales, y así mismo adjuntaremos al proyecto un vídeo en el que se pueda apreciar el real funcionamiento del código.

4.1 Reconocimiento por facciones

Vamos a mostrar imágenes que capturan la figura que el código nos devuelve con el reconocimiento de las facciones a tiempo real. Para poder apreciar con detenimiento el resultado, para cada una de estas muestras hemos anulado la impresión por pantalla de los valores de todas las facciones excepto de la que queremos mostrar en cada momento, de manera que vamos a presentar 6 imágenes con los seis reconocimientos estudiados.

En cada una de ellas podremos ver la nube de puntos que el programa crea alrededor de la cara del actor, que en este caso he sido yo misma, una caja que encuadra la cara y cinco puntos rojos en los cinco lugares de referencia que hemos establecido (ojos, nariz y comisuras de la boca).

4.1.1 Boca

El resultado que se imprime por pantalla con respecto a la boca nos dice si reconoce la boca como abierta o como cerrada. Como ya sabemos, esta información numérica se almacena en *valores (5)* en la clase *imagen,* y su contenido será 0, 1, 2 ó 3 según la posición de la boca sea desconocida, abierta, quizás cerrada o cerrada, respectivamente. A continuación vemos los dos casos que nos proveen información útil: boca abierta y boca cerrada.

a) Boca abierta

b) Boca cerrada

Figura 4-1. Reconocimiento de la apertura de la boca

4.1.2 Ojo izquierdo

La información que podemos observar por pantalla sobre el ojo izquierdo es si está abierto o cerrado o, igual que en el caso de la boca, si la posición del ojo es desconocida o 'quizás cerrado' (está probablemente cerrado pero por poca claridad en el momento de captura del frame, el código no puede asegurarlo totalmente). Este valor (0,1,2 ó 3) lo encontramos en *valores (2)* en la clase *imagen*.

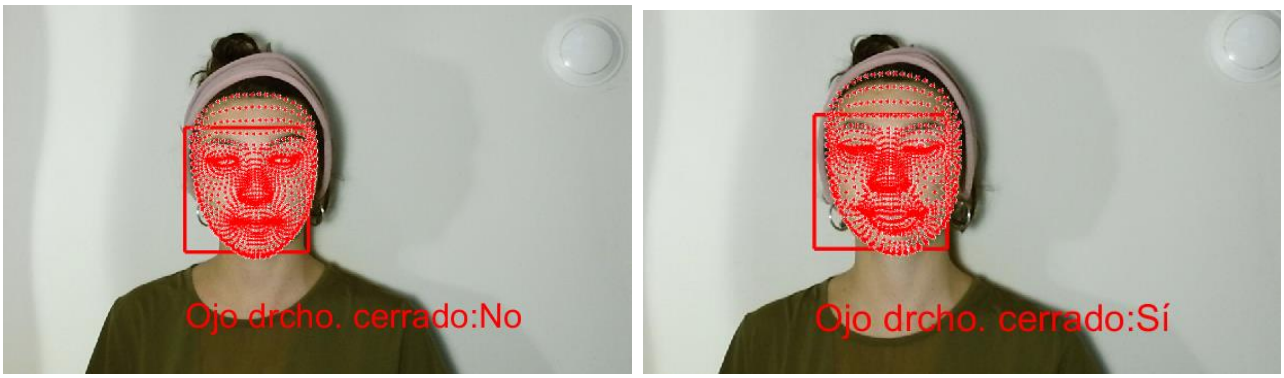
Sin embargo, ya sabemos que el código también estudia la altura a la que se encuentra la pupila (*valores (1)*), para determinar no solo si el ojo está abierto o cerrado, sino también si está mirando al frente, hacia arriba o hacia abajo, en caso de que esté abierto. El valor numérico de la altura de la pupila no está codificado para salir por pantalla, ya que no es información tan útil para el actor, pero evidentemente se procesa de la misma manera y el robot realiza los movimientos exactamente de la misma forma. Es por esto que en las imágenes veremos únicamente el reconocimiento de si el ojo izquierdo está abierto o cerrado.

a) Ojo abierto b) Ojo cerrado

Figura 4-2. Reconocimiento de la apertura del ojo izquierdo

4.1.3 Ojo derecho

Con el ojo derecho encontramos un caso análogo al anterior. La información en este caso se almacena en *valores (*4), y la correspondiente a la dirección de la mirada del ojo derecho (arriba, al frente, abajo) en *valores (3)*. En las imágenes podemos observar dicho reconocimiento.



a) Ojo abierto b) Ojo cerrado

Figura 4-3. Reconocimiento de la apertura del ojo derecho

4.1.4 Ángulo de giro de la cabeza

Este apartado estudia la orientación de la cara expresada en uno de los ángulos de Euler. Como ya se explicó en el apartado 3.3.4, los límites que hemos estudiado experimentalmente nos dicen que si el ángulo analizado por el código se encuentra entre [-20,20], la cabeza se encuentra mirando al frente (teniendo en cuenta distintas perspectivas; no siempre estará a 0 aunque esté mirando al frente). A partir de estos valores límites, el robot entenderá que la cabeza está girada hacia un lado o hacia otro. Esta información se almacena en el elemento *valores (6)* de la clase *imagen*.

En las imágenes de ejemplo podemos observar cómo al girar la cabeza hacia un lado o hacia otro, el código es capaz de reconocerlo y traducirlo numéricamente de forma correcta.

a) Cabeza hacia la izquierda

b) Cabeza hacia la derecha

c) Cabeza al frente

Figura 4-4. Reconocimiento de ángulo de giro de la cabeza

4.1.5 Ceja izquierda

El estudio de las cejas es más sensible que el del resto de las facciones. Nuestro código nos devuelve valores numéricos que se traducen a la que será la posición de la ceja: levantada, horizontal o baja. Esta información la encontramos en *valores (7)* para la ceja izquierda. Los límites que determinan la posición de la ceja han sido analizados experimentalmente, y varían según estemos procesando la ceja izquierda o la ceja derecha, como pudimos ver en el apartado 3.3.5.

En las tres imágenes inferiores observamos el análisis de la ceja en las tres posiciones diferentes que nuestro código reconoce: ceja alta, baja y horizontal

a) Ceja alta

b) Ceja baja

c) Ceja en posición horizontal

Figura 4-5. Reconocimiento de la altura de la ceja izquierda

4.1.6 Ceja derecha

El estudio de la ceja derecha es análogo al anterior, con la diferencia de que el resultado numérico obtenido esta vez se almacena en el elemento valores (8). Además, los límites numéricos que hemos establecido tras hacer un estudio experimental del procesamiento de las tres posiciones de las cejas, varían ligeramente con respecto a los límites para la ceja izquierda.

En la figura inferior podemos observar nuevamente la ceja derecha en las tres posiciones reconocidas: alta, baja y horizontal.

a) Ceja alta

b) Ceja baja

c) Ceja en posición horizontal

Figura 4-6. Reconocimiento de la altura de la ceja derecha

4.2 Procesamiento completo de reconocimiento facial

Una vez se ha mostrado que el código es capaz de identificar y procesar cada una de las facciones por separado, consideramos importante mostrar el procesamiento de expresiones faciales completas. Es decir, partiendo de la cara del actor, pasar por la figura que nos devuelve Matlab mostrando los valores numéricos y de ahí, finalmente, a la expresión del robot. Vamos a adjuntar también una visualización de los ocho canales en el momento de imitación de cada expresión, para poder visualizar gráficamente los valores -1, 0 y 1 de los que hemos estado hablando en los apartados anteriores.

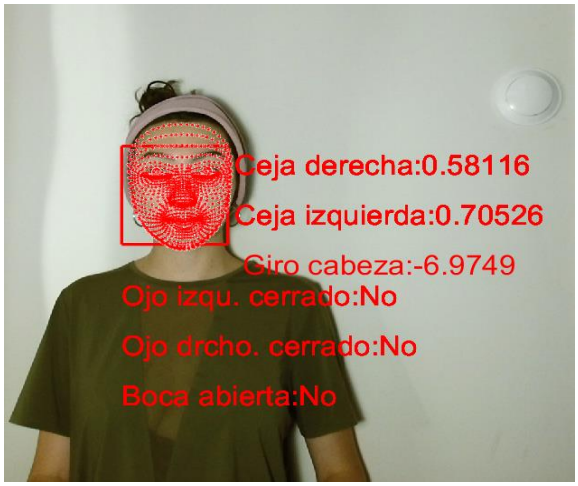
Hemos decidido mostrar dicho proceso completo en cuatro casuísticas diferentes; es decir, con cuatro expresiones faciales totalmente diferentes, para demostrar la respuesta del código ante situaciones distintas.

4.2.1 Expresión 1

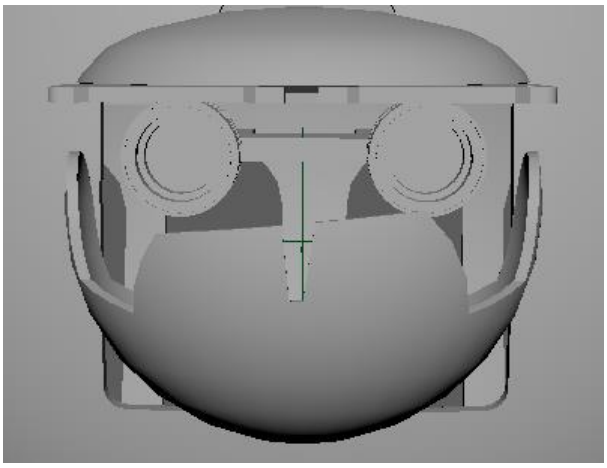
En la primera expresión se produce la identificación de una cara neutra. Como vemos en la figura 4-7 b), se reconocen claramente ambos ojos abiertos y la boca cerrada. El giro de la cabeza está dentro del rango que se considera como 'cabeza al frente' y las cejas igual, si bien ya hemos comentado anteriormente que la detección numérica de las cejas es excesivamente sensible.

Podemos comprobar en 4-7 d) que al robot le han llegado todos los valores a 0 (están en el centro del rango de movimiento, que va de [-1,1]), por lo que la expresión del mismo es absolutamente neutra.

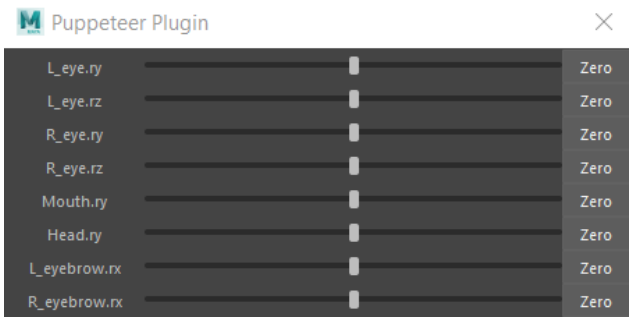
a) Expresión original del usuario



b) Reconocimiento por facciones



c) Expresión de imitación del robot

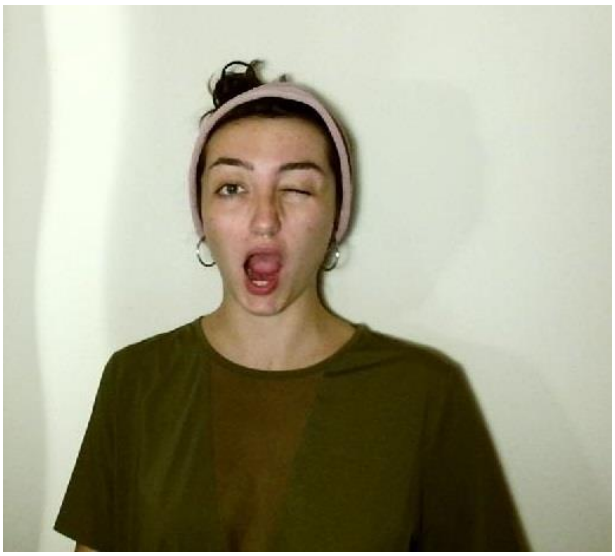


d) Posición de los canales en el momento de la imitación

Figura 4-7. Representación completa de la expresión 1

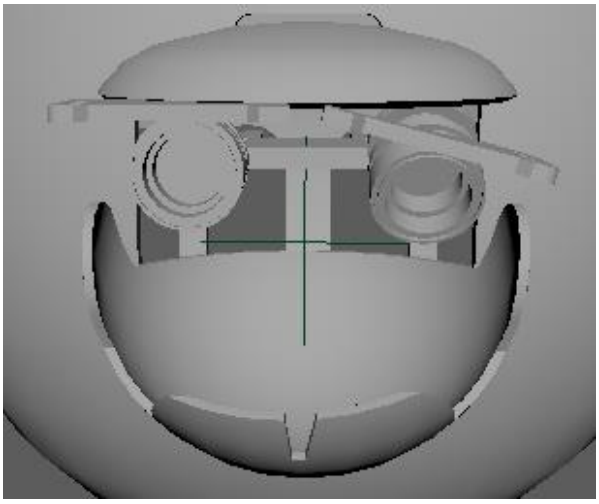
4.2.2 Expresión 2

En esta segunda expresión ponemos a prueba la efectividad del código realizando movimientos faciales más concretos. Como vemos en 4-8 b), el ojo izquierdo se mantiene normal mientras que el ojo derecho se identifica como cerrado. Esto se traduce en valores numéricos que cambian la posición de los canales, provocando así el movimiento del ojo y ceja derechos. Igualmente, la cabeza se encuentra dentro del rango de normalidad y la boca se detecta como abierta, por lo que podemos observar que el canal 5 (el que corresponde a la boca) está totalmente abierto (es decir, en valor -1) y que el canal 6 se mantiene en el centro.

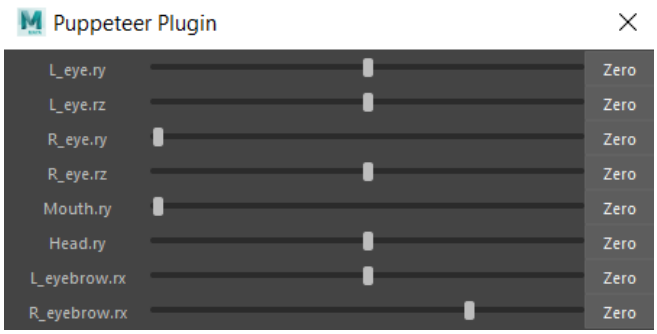


a) Expresión original del usuario

b) Reconocimiento por facciones



c) Expresión de imitación del robot



d) Posición de los canales en el momento de la imitación

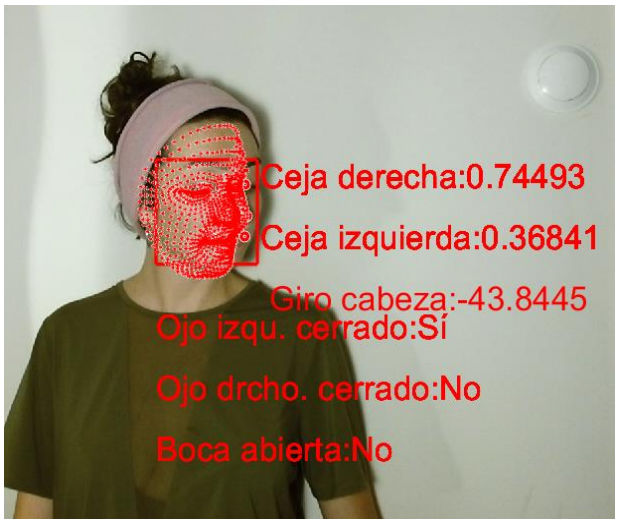
Figura 4-8. Representación completa de la expresión 2

4.2.3 Expresión 3

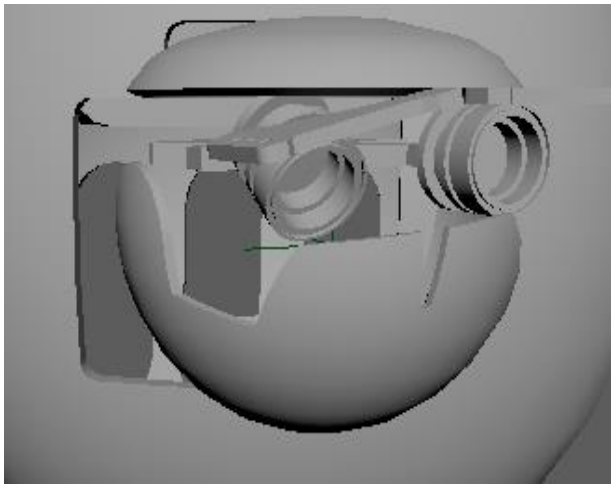
El ejemplo 3 introduce una nueva comprobación de la efectividad del código: giramos la cabeza hacia la derecha y, a su vez, el ojo izquierdo está cerrado. Comprobamos en 4-9 b) que se reconoce el giro de la cabeza, ya que el ángulo encontrado es menor de -20, que se estableció como el ángulo límite para la identificación de cabeza girada. Así mismo, boca y ojo derecho se identifican correctamente, mientras que el ojo izquierdo también se reconoce como cerrado.

Una vez más vemos estos valores traducidos a los canales (todos valen -1, 0 ó 1, excepto las cejas que al cerrarse obtienen el valor de ±0.5, según de qué ceja se trate (esto ya se explicó con detalle en apartados previos). El robot toma la información de los canales y, como resultado de todo el proceso, realiza la imitación de la cara del usuario.

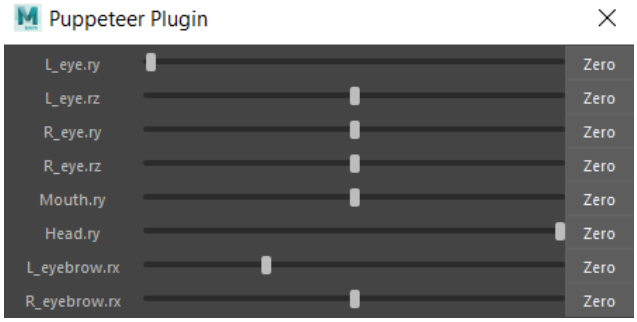
a) Expresión original del usuario



b) Reconocimiento por facciones



c) Expresión de imitación del robot

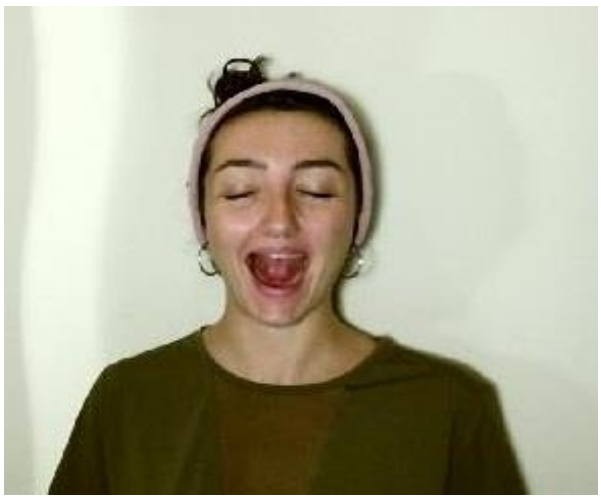


d) Posición de los canales en el momento de la imitación

Figura 4-9. Representación completa de la expresión 3

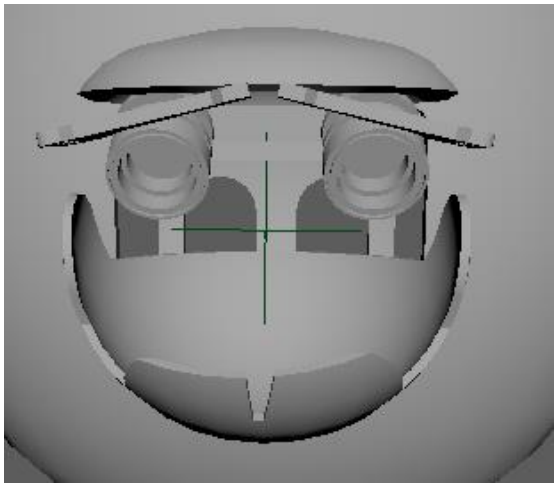
4.2.4 Expresión 4

La última expresión ejemplificada trata de llevar al límite todas las propiedades que nuestro sistema es capaz de reconocer: ambos ojos cerrados, cejas bajas y boca abierta. Podemos comprobar que desde Matlab se reconocen todos los valores correctamente (figura 4-10 b), y que esto se ve traducido en los canales del robot en Maya (figura 4-10 d), por lo que finalmente el robot realiza la imitación exacta de la expresión facial del usuario.

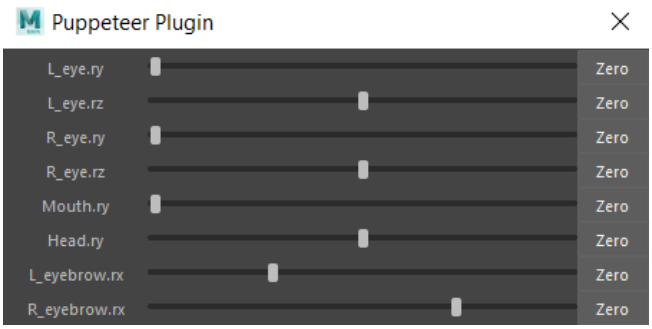


a) Expresión original del usuario

b) Reconocimiento por facciones



c) Expresión de imitación del robot



d) Posición de los canales en el momento de la imitación

Figura 4-10. Representación completa de la expresión 4

4.3 Análisis

Los ejemplos plasmados más arriba conforman la parte de la totalidad de los resultados que podemos mostrar en este formato. Sin embargo, la singularidad del proyecto no sólo reside en la capacidad de realizar correctamente estos procesamientos, sino en hacerlo en un formato de grabación de imágenes a tiempo real, que es algo que por su naturaleza no podemos plasmar en una memoria de este tipo.

No obstante, podemos decir que el conjunto de resultados que obtenidos del proyecto responden con éxito al objetivo inicial de la investigación, ya que se ha conseguido exactamente lo que se buscó en un principio. Si bien es cierto que realicé una ardua labor de investigación previa que me proporcionó una visión extensa de lo que ya existía, la realidad es que el código generado es totalmente genuino y original de este proyecto, y que ha demostrado tener unos resultados muy positivos.

La documentación previa me permitió leer y procesar mucho código de procesamiento de imágenes en Matlab. Sin embargo, ninguno de ellos conseguía obtener resultados exactos o concretos sobre reconocimiento facial. Primeramente, porque no hacían análisis de imágenes a tiempo real sino de imágenes sueltas, lo que hacía de estos códigos algo denso, lento y difícil de aplicar a un objetivo como el nuestro. Pero además, la grandísima mayoría de ellos trataban de obtener buenos resultados utilizando la webcam que traen los portátiles incorporadas o, en algunos casos, webcams externas instaladas en el ordenador. Estas cámaras no tienen las propiedades de análisis de profundidad, infrarrojo o de detección corporal con laa que cuenta la cámara que nosotros hemos empleado, por lo que aún si se consiguiera implementar estos códigos a tiempo real, no hubieran proporcionado ninguna solución implementable para nuestro objetivo.

5 CONCLUSIONES

El proyecto de control de robot animatrónico mediante Visión Artificial tuvo como cometido inicial conseguir que un robot físico, que nos fue cedido para el desarrollo del proyecto por parte de la empresa *Puppeteer Technologies*, representara a tiempo real las expresiones faciales de un actor. Para ello he podido trabajar con distintos softwares de grabación, tratamiento y envío de información que han sido descritos con total detalle en apartados previos a este. Como síntesis de los resultados del proyecto una vez finalizado, podemos decir que el código creado ha cumplido con éxito los objetivos propuestos y que hemos conseguido abrir un nuevo camino de estudio e investigación en la rama de la animatrónica en la Universidad, aun con las dificultades encontradas, como detallaremos a continuación.

Antes de describir los impedimentos con los que hemos tenido que lidiar y las posibles líneas de trabajo futuro, voy a mencionar algunas características del proyecto que merecen ser citadas individualmente por su aportación positiva o innovadora al mismo.

Primeramente, creo que una de las mayores atracciones que tiene la identificación de imagen de este proyecto es la particular nube de puntos que se crea alrededor de la cara del actor. Este modelo está formado por una malla de 1347 puntos alrededor de la cara. Lo que la hace destacar es que estos puntos crean un modelo tridimensional que, a tiempo real, sigue a la cara del actor por donde se mueva y realice el movimiento que realice, incluso si la cara deja de verse completamente en la pantalla. Los pocos códigos encontrados en la investigación previa a la realización del proyecto que eran capaces de crear nubes de puntos alrededor de la cara del actor, lo hacían procesando una sola imagen, y además no podían hacer una detección de la cara de forma tridimensional, haciendo así de nuestra nube de puntos una característica positivamente llamativa de nuestro proyecto.

Tenemos que destacar en este apartado también la eficacia y la facilidad del robot que se nos ha proporcionado. Tanto el robot físico como la versión en Maya, que es una copia digital exacta del mismo, han aportado una grandísima fluidez a todo nuestro trabajo desde el principio. Una vez realizado un estado del arte sobre el tema pude observar cómo los robots diseñados para la grabación de grandes películas de animación son, por regla, excesivamente expresivos en sus gestualidades, sobre todo las faciales, hasta un punto que podría ser calificado como 'poco real'. Una expresión que de primeras pudiera resultar exagerada es justo lo que se busca a la hora de diseñar un robot con este cometido. Es por esto que el nuestro capta totalmente esa esencia, teniendo un rango de expresión para cada una de las facciones un poco más amplio del que probablemente ninguno de nosotros le hubiera asignado de primeras, situando así a nuestro trabajo un poco más cerca de lo que podría llegar a ser en un futuro un proyecto de grabación real.

Así mismo, es importante realzar el hecho de que todo el código de Matlab (en total se necesitan más de mil líneas de código para procesar nuestro proyecto) está desarrollado en Programación Orientada a Objetos (POO). Como ya describimos anteriormente, se trata de un paradigma de programación que viene a innovar la forma de obtener resultados, basándose en técnicas algo más complejas que la programación estructurada como son la herencia, la abstracción, la encapsulación, etc. No obstante, si tuviéramos que buscar alguna desventaja a la POO podríamos decir que requiere un cambio en la manera de pensar desde la programación tradicional a la que estamos acostumbradas, obligando así a un aprendizaje y entrenamiento previo.

Antes de iniciar este proyecto yo nunca había programado de este modo, de manera que cuando supe que el desarrollo de mi TFG se basaría en esta forma de programar decidí inscribirme a un curso online, impartido en inglés por el *Indian Institute of Technology* (IIT) de Bombay a través de la plataforma edX (23). Tras realizar este curso, titulado *Object-Oriented Programming in C++,* pude crear el código con mucha más normalidad, y de ahí que haya sido posible que el núcleo de nuestro código lo conformen tres clases estructuradas jerárquicamente, como ya hemos visto en apartados anteriores.

5. 1 Dificultades encontradas

No obstante, no sería cierto afirmar que no se han encontrado grandes dificultades a lo largo del desarrollo de este proyecto. Antes de meterme de lleno en un tema tan ajeno como era la visión artificial para la animatrónica, tuve que hacer un estado del arte que me permitió leer, entender y probar mucho código para Matlab de temas que se engloban bajo el título de 'Reconocimiento Facial' (o *Facial Mocap*, que es como di con la mayor parte de la información).

Al inicio de mi investigación, la línea que pensaba que seguiría era la de hacer un tratamiento de la imagen de una cara con Matlab siguiendo métodos tradicionales (suavizado, uso de histogramas, curvas tonales, etc.). Así, fui dando con diferentes proyectos que trataban el procesamiento de imágenes de una manera o de otra. Los códigos que estudié en esta primera etapa de desarrollo del proyecto fueron muchos y muy diversos, y todos ellos parecían una gran base para avanzar en el mismo. Los dos proyectos que voy a mencionar a continuación sirven como ejemplo de la dirección que llevaba mi búsqueda, aun siendo sólo dos de los muchos que estudié.

El primero de ellos es un proyecto que trata de detectar la somnolencia en conductores de camiones a través del tratamiento de imágenes en Matlab (24). Para ello, se hacía un análisis por balance de blancos del nivel de apertura de los ojos, y en el momento en que se detectaba que estos estaban suficientemente cerrados, saltaba la alarma para despertar al conductor y que este dejara de conducir. De este proyecto me pareció muy interesante, de primeras, el estudio del nivel de apertura de los ojos, pensando que podría implementarlo en mi proyecto para estudiar cuándo mi actor abriera o cerrara los ojos, y así pasárselo al robot.

El segundo proyecto que voy a mencionar fue realizado por un grupo de estudiantes investigadores de una universidad brasileña. Su objetivo era localizar unos determinados puntos de referencia faciales (25) a partir de una foto de la cara del actor. Mi idea original era, a partir de este código, realizar un rastreo de los movimientos de cada uno de esos puntos para así conseguir formar la trayectoria facial del actor y poder pasársela al robot.

Sin embargo, todas las expectativas que tenía con estos proyectos fueron desechándose conforme fui probando sus códigos por dos motivos principales. El primero de ellos tiene que ver con el tratamiento de imágenes a tiempo real, que como venimos diciendo es una de las singularidades de nuestro proyecto. La mayoría de códigos realizaban un tratamiento de imágenes por píxeles usando distintos métodos, por lo que el análisis de una sola imagen conllevaba demasiado tiempo y no hacía factible el procesamiento de imágenes a tiempo real. No obstante, si este procesamiento de imágenes hubiese sido válido, hubiera intentado encontrar la manera de aligerarlo y hacerlo factible a tiempo real. Pero aquí llega el segundo gran impedimento. Ninguno de los códigos investigados para Matlab hacía uso de una cámara que no fuera la cámara incorporada al portátil o, en su defecto, una webcam instalada en el PC. De tal manera que las propiedades que realmente han hecho que nuestro proyecto funcione, como son la detección de profundidad, la radiación infrarroja o la localización simultánea de puntos en el cuerpo, no están disponibles para los códigos estudiados, provocando así que acabara desechándolos como punto de partida de nuestro proyecto.

Una vez entendí que el dispositivo empleado para la grabación era totalmente decisivo y conseguí, además, realizarla a tiempo real, los dos grandes obstáculos que había encontrado en el resto de proyectos habrían sido salvados en el mío, sabiendo ahora que partía de una base sólida que me permitiría alcanzar todos los objetivos propuestos.

5.2 Valoración personal

Pese a todos los obstáculos y dificultades encontradas, podemos decir que el conjunto de resultados que finalmente se han obtenido del proyecto responden con éxito al objetivo inicial de la investigación, ya que se ha conseguido exactamente lo que se buscó en un principio. Si bien es cierto que realicé una ardua labor de investigación previa que me proporcionó una visión extensa de lo que ya existía, la realidad es que el código generado es totalmente genuino y original para este proyecto, y que ha demostrado tener unos resultados muy positivos.

Fue a partir del momento en que conseguí hacerme con el dispositivo de grabación adecuado cuando el proyecto comenzó a tomar forma; poco a poco fue creciendo y superando obstáculos, hasta que se completó. A día de hoy puedo afirmar que los resultados que se han obtenido fueron el objetivo inicial de la investigación que se nos propuso. Con mucho detalle fui creando, corrigiendo y reescribiendo el código hasta que conseguí que el

procesamiento de imágenes y la red de comunicación y envío de datos entre los distintos elementos del proyecto fueran rápidas, funcionales y eficaces, por lo que me siento satisfecha del trabajo realizado.

5.2 Líneas de trabajo futuro

Sin embargo, es evidente que cualquier proyecto de investigación tiene puntos de futuras mejoras, y este no será diferente. El principal punto en que cabría profundizar sería la velocidad de respuesta. Pese a que estamos trabajando con una grabación de imágenes a tiempo real y el tiempo que transcurre desde que se capta un *frame* hasta que el robot copia la expresión facial es muy corto, sabemos que estamos trabajando con Matlab y que su mayor fuerte no es el trabajo con funciones a tiempo real. De hecho, es posible que sea este el motivo por el cual la gran parte de los proyectos de Facial Tracking estudiados no intentaban realizar este procesamiento a tiempo real: Matlab no es la mejor herramienta para ello. Es por eso que dentro de las posibilidades reales de este programa, el procesamiento que realizamos es especialmente rápido y eficiente, también gracias al empleo de POO como forma de programación.

De esta manera, aquí se define una primera línea de trabajo futuro: mejorar el *frame rate* de nuestro proyecto, quizás traduciéndolo a otro lenguaje de programación, de manera que la imitación de las expresiones faciales sea absolutamente instantánea.

Además, cabe mencionar que otra posible 'mejora' que podría tener este proyecto sería la implementación del código en el robot físico. Como ya explicamos al inicio de la memoria, la idea original era pasar la información desde *Puppeteer Technologies* al robot físico, pero dado que el proyecto se ha desarrollado casi en totalidad durante la cuarentena por el Covid-19, tanto el robot físico como el resto de elementos de conexión necesarios estaban en la universidad en el momento del inicio de la pandemia y no hemos podido acceder a ellos. Por suerte, tuvimos la opción de modificar el final del proyecto gracias a la copia virtual de nuestro robot y todo ha concluido con éxito. Pero no deja de ser una línea de trabajo que se retomará en cuanto tengamos acceso al material necesario.

Centrándonos en el código en sí mismo, también hay varios detalles definidos a los que se les podría sacar partido en una futura profundización de este trabajo. Primeramente, como explicamos más arriba, el movimiento del robot se produce porque Matlab envía datos numéricos a Puppeteer, y de ahí llegan a los canales del robot. Nuestro proyecto ha requerido una investigación previa y finalmente supone una base de trabajo para futuras profundizaciones en el tema. Es por ello que no nos hemos dedicado a estudiar todas las alternativas de cualquier movimiento, sino a crear los propios movimientos y las interfaces que permiten su envío, lectura y realización. Para cada canal de Maya, los únicos valores que hemos estudiado han sido tres: -1,0 ó 1. Por ejemplo, en el caso del giro de la cabeza, la cabeza gira hacia la izquierda, al frente o hacia la derecha. Una futura ampliación del proyecto podría, con total facilidad, crear más límites numéricos y definir así más posiciones de giro de la cabeza. De esta manera, los canales de Maya podrían recibir diferentes valores numéricos más allá del -1, 0 ó 1. Por ejemplo, se podrían enviar valores por -1, -0.75, -0.5, -0.25, 0, 0.25, y así sucesivamente, creando nueve posiciones de giro de cabeza en lugar de las tres que existen actualmente, aprovechando todo el margen de movimiento que tiene nuestro robot.

Con respecto a posibles ampliaciones de código, también vamos a mencionar la opción de que nuestra clase *comunic*, cuya finalidad ya sabemos que es la de crear la comunicación por OSC con *Puppeteer*, fuera capaz no solo de enviar esta información, sino también de recibirla. En esta cuestión hemos ahondado un poco más y hemos llegado a crear el código que se debería añadir a la clase *comunic* para que fuera capaz de recibir paquetes de datos por OSC (22). Este código ha sido ejecutado y se ha comprobado su correcto funcionamiento; se encuentra adjunto en el Anexo 3 y comentado como función *receiver* dentro de la clase *comunic* (sería la posible función que realizaría lo que estamos comentando, análoga a la actual *sender*). Aquí se abre otra posible línea de trabajo futuro en la que el robot físico realizara, quizás, movimientos dirigidos telemáticamente, y nuestro proyecto fuera capaz de traducirlos numéricamente y analizarlos mediante gráficas, programas de vídeo, etcétera.

Por último, quiero concluir la enumeración de posibles líneas de trabajo futuro mencionando la AzureKinect DK. Lo cierto es que la fabricación de Kinect para Windows se ha descontinuado, como ya anunciamos al principio de esta memoria. La próxima generación de sensores de profundidad que marcarán la diferencia de Microsoft se llama Azure Kinect DK (26). Se trata de un kit para desarrolladores con sensores avanzados de

Inteligencia Artificial para modelos de voz e imágenes sofisticados para equipos. De tal manera que una vez estos sensores estén a nuestra disposición sería un trabajo muy interesante integrarlos en nuestro proyecto con las diferencias y, consecuentemente, las mejoras que aportará al mismo.

Para finalizar con las conclusiones sacadas de todo el trabajo realizado, quiero poner sobre la mesa las aplicaciones cercanas y reales que este tendrá. Por una parte, la empresa *Puppeteer* estuvo desde el primer momento interesada en colaborar con nosotros porque hemos tratado un tema muy atractivo para la industria del cine de animación. De hecho, están muy satisfechos con el resultado y ven la opción de aplicarlo a algún futuro proyecto y tener la posibilidad de trabajar juntos.

Más allá de eso, este proyecto era necesario para sembrar en nuestra facultad la semilla de la animatrónica. Es una disciplina de la robótica sobre la que no se había realizado ningún trabajo o investigación previa en nuestra universidad, y de hace pocos años aquí se ha convertido en una ciencia potencialmente interesante en el mundo de la automática. Así, coetáneamente a este TFG se ha realizado otro proyecto de investigación en domótica con Python, cuyo software se espera que se complete adaptándose al robot vía Puppeteer para poder realizar el procesamiento total en un futuro no muy lejano. De esta forma, tendremos los dos lenguajes de programación más estudiados académicamente cubiertos. Cuando esto se haya completado, todo este desarrollo será alojado en una página web como software de distribución libre, promoviendo así el uso de la animatrónica como plataforma de enseñanza para el control y la robótica.

REFERENCIAS

1. CONTAVAL: La Visión Artificial. [online]. 2016. Available from: https://www.contaval.es/que-es-la-vision-artificial-y-para-que-sirve/
2. Animatrónica - Wikipedia, la enciclopedia libre. [online]. 2020. Available from: https://es.wikipedia.org/wiki/Animatrónica
3. Captura de movimiento - Wikipedia, la enciclopedia libre. [online]. 2020. Available from: https://es.wikipedia.org/wiki/Captura_de_movimiento
4. MERISTATION MAGAZINE, S.L. Captura de movimiento: historia y análisis en los videojuegos. [online]. 2019. Available from: https://as.com/meristation/2017/07/27/reportajes/1501135200_167918.htmlLa captura de movimiento ha cambiado las producciones triple A y la animación de los personajes creados.
5. STATISTA. • Industria de animación: valor total del mercado 2017-2020 | Statista. [online]. 2019. Available from: https://es.statista.com/estadisticas/942083/tamano-de-la-industria-de-animacion-mundial/
6. DANIEL QUINTERO Y JAVIER CORONILLA. Puppeteer Technologies. *2020* [online]. Available from: https://www.puppeteer.tech/
7. Hay Vida en Martes: Animatrónicos, los robots del cine | Espacio Fundación Telefónica. [online]. 2020. Available from: https://espacio.fundaciontelefonica.com/evento/hay-vida-en-martes-animatronicos-los-robots-del-cine/
8. TERVEN, Juan R. and CÓRDOVA-ESPARZA, Diana M. Kin2. A Kinect 2 toolbox for MATLAB. *Science of Computer Programming*. 15 November 2016. Vol. 130, p. 97–106. DOI 10.1016/j.scico.2016.05.009. This paper introduces Kin2, a Kinect 2 toolbox for MATLAB. This toolbox encapsulates most of the Kinect for Windows SDK 2.0 functionality in a single class with high-level methods. The toolbox is written mostly in C++ with MATLAB Mex functions providing access to color, depth, infrared, and body index frames; coordinate mapping capabilities; real-time six-body tracking with 25 joints and hands states; face and high-definition face processing; and real-time 3D reconstruction. We showed that the performance of a Kin2 application decreases 30% on average with respect to a native C++ application, however, Kin reduces in one order of magnitude the average code length yielding a significant reduction in development time for prototyping and research.
9. DUQUE, Edwin. Qué es el Microsoft Kinect? | Toward a brand-NUI World. [online]. 2015. Available from: https://edwinnui.wordpress.com/2015/02/03/qu-es-el-microsoft-kinect/
10. Kinect - Wikipedia, la enciclopedia libre. [online]. 2020. Available from: https://es.wikipedia.org/wiki/Kinect#Kinect_2.0
11. PASCUAL, Juan Antonio. Así es Kinect 2.0 para Windows en PC | Tecnología - ComputerHoy.com. [online]. 2014. Available from: https://computerhoy.com/noticias/hardware/asi-es-kinect-20-windows-pc-10937
12. WINDOWS. Microsoft libera Kinect SDK 2.0 y nuevo kit adaptador - News Center Latinoamérica. [online]. 2014. Available from: https://news.microsoft.com/es-xl/microsoft-libera-kinect-sdk-2-0-y-nuevo-kit-adaptador/
13. Adaptador Kinect, Adapter USB 3.0. [online]. 2020. Available from: https://gorilageek.com/producto/peaklead-es-kinect-usb-controller-tarjetas-de-puertos-tarjetas-de-puerto-usb/

14. Microsoft lanza el Software de Desarrollo de Kinect para Windows (SDK), una beta para desarrolladores inquietos. [online]. 2014. Available from: https://www.zonamovilidad.es/noticia/2799/tecnologia/microsoft-lanza-el-software-de-desarrollo-de-kinect-para-windows-sdk-una-beta-para-desarrolladores-inquietos.html

15. MICROSOFT. FaceShapeAnimations Enumeration. [online]. 2014. Available from: https://docs.microsoft.com/en-us/previous-versions/windows/kinect/dn791602(v=ieb.10)?redirectedfrom=MSDN

16. Ventajas de la Programación Orientada a Objetos. [online]. Available from: https://www.ciberaula.com/cursos/java/ventajas_poo.php

17. MATHWORKS. Funciones de archivos MEX - MATLAB & Simulink - MathWorks España. [online]. Available from: https://es.mathworks.com/help/matlab/call-mex-file-functions.html

18. HighDetailFacePoints Enumeration | Microsoft Docs. [online]. 2014. Available from: https://docs.microsoft.com/en-us/previous-versions/windows/kinect/dn791778(v=ieb.10)?redirectedfrom=MSDN

19. LARIO, Sergi. Introducción al protocolo de comunicación Open Sound Control. *Mosaic* [online]. 2019. No. 170. DOI 10.7238/m.n170.1924. Available from: https://mosaic.uoc.edu/2019/04/03/introduccion-al-protocolo-de-comunicacion-open-sound-control/Este artículo presenta una introducción al protocolo de comunicación Open Sound Control incorporado en la actualidad en diferentes herramientas multimedia para la producción de instalaciones interactivas audiovisuales. Está basado en un sistema abierto de mensajes por la red y permite la interoperabilidad entre distintos dispositivos y software, facilitando la sincronización en tiempo real de procesos entre ellos.

20. opensoundcontrol.org | an Enabling Encoding for Media Applications. [online]. Available from: http://opensoundcontrol.org/

21. Autodesk Maya - Wikipedia, la enciclopedia libre. [online]. Available from: https://es.wikipedia.org/wiki/Autodesk_Maya#Interfaz_gráfica

22. MARIJNISSEN, MARK. Send Open Sound Control (OSC) Messages - File Exchange - MATLAB Central. [online]. 2011. Available from: https://es.mathworks.com/matlabcentral/fileexchange/31400-send-open-sound-control-osc-messages

23. edX | Cursos online gratis de Harvard, MIT y más | edX. [online]. Available from: https://www.edx.org/es

24. Drowsiness Detection using a Binary SVM Classifier - File Exchange - MATLAB Central. [online]. Available from: https://es.mathworks.com/matlabcentral/fileexchange/55152-drowsiness-detection-using-a-binary-svm-classifier?s_tid=mwa_osa_a

25. Facial Landmarks - File Exchange - MATLAB Central. [online]. Available from: https://es.mathworks.com/matlabcentral/fileexchange/47713-facial-landmarks

26. Kinect: desarrollo de aplicaciones de Windows. [online]. Available from: https://developer.microsoft.com/es-es/windows/kinect/

ANEXOS

A – Anexo 1. Función drawFaces de la clase kin2 tras la modificación realizada

```
function
data_faces=drawFaces(this,handle,faces,pointsSize,displayText,fontSize)
    if nargin < 6
                fontSize = 20;
    end

    numFaces = size(faces,2);
     data_faces=zeros(1,5);

    % Draw each body
    for i=1:numFaces

    % Draw the facial landmarks


viscircles(handle,faces(i).FacePoints',ones(5,1)*pointsSize,'EdgeColor',
this.bodyColors(i));

    data_faces(1)=(faces.FacePoints(2)*100)/faces.FacePoints(6);
%Posición   en el eje y de la pupila del ojo izqu.
    data_faces(2)=faces.FaceProperties(4); %Ojo_izqu cerrado?
    data_faces(3)=(faces.FacePoints(4)*100)/faces.FacePoints(6);
%Posición en el eje y de la pupila del ojo drcho.
    data_faces(4)=faces.FaceProperties(5); %Ojo_drcho cerrado?
    data_faces(5)=faces(i).FaceProperties(6); %Boca

    % Draw the rectangle
    left = faces(i).FaceBox(1);
    top = faces(i).FaceBox(2);
    right = faces(i).FaceBox(3);
    bottom = faces(i).FaceBox(4);

    % Top line
    line([left right],[top top],'Color',this.bodyColors(i), ...
                    'LineWidth',2,'Parent',handle);
    % Bottom line
    line([left right],[bottom bottom],'Color', this.bodyColors(i),
'LineWidth',2,'Parent',handle);
    % Left line
    line([left left],[top bottom],'Color',this.bodyColors(i), ...
                    'LineWidth',2,'Parent',handle);
    % Right line
    line([right right],[top bottom],'Color',this.bodyColors(i), ...
                    'LineWidth',2,'Parent',handle);

    if displayText

       % Display face information

        for j=1:length(this.faceProperties)
```

```
        %MOSTRAR INFO OJO_IZQU (Abierto/Cerrado)
        yoffset = 3*fontSize*1.5;
        p = faces(i).FaceProperties(4);
        strp = this.decodeFaceProperties(p);
        str = strcat(this.faceProperties(4),':',strp);
        text(left,bottom + yoffset ,str, ...
        'Color',this.bodyColors(i),'FontSize',fontSize, ...
        'FontUnits','pixels','Parent',handle);

        %MOSTRAR INFO OJO_DRCHO (Abierto/Cerrado)
        yoffset = 6*fontSize*1.5;
        p = faces(i).FaceProperties(5);
        strp = this.decodeFaceProperties(p);
        str = strcat(this.faceProperties(5),':',strp);
        text(left,bottom + yoffset ,str, ...
        'Color',this.bodyColors(i),'FontSize',fontSize, ...
        'FontUnits','pixels','Parent',handle);

        %MOSTRAR INFO BOCA
        yoffset = 9*fontSize*1.5;
        p = faces(i).FaceProperties(6);
        strp = this.decodeFaceProperties(p);
        str = strcat(this.faceProperties(6),':',strp);
        text(left,bottom + yoffset ,str, ...
        'Color',this.bodyColors(i),'FontSize',fontSize, ...
        'FontUnits','pixels','Parent',handle);

        end

      end

     end

end
```

B – Anexo 2. Función drawHDFaces de la clase kin2 tras la modificación realizada

```
function
data_HDfaces=drawHDFaces(this,handle,faces,displayPoints,displayText,font
Size)
      if nargin < 6
         fontSize = 20;
      end

      numFaces = size(faces,2);

      % Draw each body

      for i=1:numFaces

        % Draw the rectangle if not showing the points

          left = faces(i).FaceBox(1);
          top = faces(i).FaceBox(2);
          right = faces(i).FaceBox(3);
          bottom = faces(i).FaceBox(4);

          if ~displayPoints

          %Top line
           line([left right],[top top],'Color',this.bodyColors(i), ...
           'LineWidth',2,'Parent',handle);
          %Bottom line
           line([left right],[bottom bottom],'Color',
           this.bodyColors(i),'LineWidth',2,'Parent',handle);
          %Left line
           line([left left],[top bottom],'Color',this.bodyColors(i),
           'LineWidth',2,'Parent',handle);
          %Right line
           line([right right],[top bottom],'Color',this.bodyColors(i),
           'LineWidth',2,'Parent',handle);

          end

          if displayText

          % Display face information
            for j=1:length(this.faceAnimationUnits)

            %DISPLAY CEJA_DRCHA
             yoffset = 1*fontSize*1.5;
             p = faces(i).AnimationUnits(14);
             str =strcat(this.faceAnimationUnits(14),':',num2str(p));
                  text(right,top + yoffset ,str,
                  'Color',this.bodyColors(i),'FontSize',fontSize, ...
                  'FontUnits','pixels','Parent',handle);
```

```
            %DISPLAY CEJA_IZQU
              yoffset = 4*fontSize*1.5;
              p = faces(i).AnimationUnits(15);
              str=strcat(this.faceAnimationUnits(15),':',num2str(p));
              text(right,top + yoffset ,str, 'Color',

this.bodyColors(i),'FontSize',fontSize,'FontUnits','pixels','Parent',handl
e);
            end

            % DISPLAY YAW
              pitch = faces(i).FaceRotation(1);
              yaw = faces(i).FaceRotation(2);
              roll = faces(i).FaceRotation(3);
              str = [' Giro cabeza:' num2str(yaw)];
              yoffset = 7*fontSize*1.5;% yoffset + fontSize*1.5;
              text(right,top + yoffset ,str, ...
              'Color',this.bodyColors(i),'FontSize',fontSize, ...
              'FontUnits','pixels','Parent',handle);

          end

          % Display HD points on color image
            if displayPoints

            colorCoords=this.mapCameraPoints2Color(faces(i).FaceModel');

viscircles(handle,colorCoords,ones(1347,1)*1,'EdgeColor',
this.bodyColors(i));
            end

            data_HDfaces(1)=faces(i).FaceRotation(2);%yaw;
            data_HDfaces(2)=faces(i).AnimationUnits(15); %Ceja izquierda
            data_HDfaces(3)=faces(i).AnimationUnits(14); %Ceja derecha

     end

end
```

C – Anexo 3. Función oscsend

```
function oscsend(u,path,varargin)

    [~, ~, endian] = computer;
    littleEndian = endian == 'L';

    % Set type
    if nargin >= 2,
        types = oscstr([',' varargin{1}]);
    else
        types = oscstr(',');
    end;

    % Set args (either a matrix or varargin)
    if nargin == 3 && length(types) > 2
        args = varargin{2};
    else
        args = varargin(2:end);
    end;

    % Convert arguments to the right bytes
    data = [];
    for i=1:length(args)
        switch(types(i+1))
            case 'i'
                data = [data oscint(args{i},littleEndian)];
            case 'f'
                data = [data oscfloat(args{i},littleEndian)];
            case 's'
                data = [data oscstr(args{i})];
            case 'B'
                if args{i}
                    types(i+1) = 'T';
                else
                    types(i+1) = 'F';
                end;
            case {'N','I','T','F'}

            otherwise
                warning(['Unsupported type: ' types(i+1)]);
        end;
    end;

    %Write data to UDP

    data = [oscstr(path) types data];
    fwrite(u,data);

end
```

```
%Conversion from double to float

function float = oscfloat(float,littleEndian)
   if littleEndian
        float = typecast(swapbytes(single(float)),'uint8');
   else
        float = typecast(single(float),'uint8');
   end;
end

%Conversion to int
function int = oscint(int,littleEndian)
   if littleEndian
        int = typecast(swapbytes(int32(int)),'uint8');
   else
        int = typecast(int32(int),'uint8');
   end;
end

%Conversion to string (null-terminated, in multiples of 4 bytes)
function string = oscstr(string)
    string = [string 0 0 0 0];
    string = string(1:end-mod(length(string),4));
end
```