UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

INGENIERÍA

EN INFORMÁTICA

PROYECTO FIN DE CARRERA

MOCASYM: Sistema de captura semi-automática del movimiento para la representación de la Lengua de Signos Española

Roberto Mancebo Campos

Septiembre, 2010

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

Departamento de Informática

PROYECTO FIN DE CARRERA

Autor: Roberto Mancebo Campos
Director: Dr. Carlos González Morcillo

Septiembre, 2010.

TRIBUNAL:

Presidente:
Vocal1:
Vocal2:
Secretario:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE	VOCAL1	VOCAL2	SECRETARIO
Fdo.:	Fdo.:	Fdo.:	Fdo.:

Este documento fue compuesto con LaTeX. Imágenes generadas con OpenOffice.

Resumen

Según datos de la CNSE (Confederación Estatal de Personas Sordas) en España hay más de 100.000 personas con deficiencias auditivas profundas que emplean la Lengua de Signos Española (LSE) como medio de comunicación. Actualmente, se están desarrollando las primeras soluciones software de traducción automática a esta lengua.

El presente proyecto fin de carrera consiste en la construcción de un sistema de captura de movimiento que toma como entrada secuencias de vídeo real en el ámbito de la LSE. En los vídeos, un intérprete emplea LSE en un entorno controlado (posición de la cámara) pero sin añadir elementos activos o pasivos que faciliten la captura. Estas restricciones permiten reducir los tiempos de grabación de los signos sin necesidades de hardware de captura de movimiento específico.

Así, el objetivo principal del proyecto es la creación de una herramienta de análisis y edición de movimientos para facilitar la construcción de un diccionario de signos. Esta salida será en un formato estándar basado en XML que podrá ser utilizado en distintos avatares virtuales. En concreto, se plantea su utilización en el ámbito del proyecto GANAS (Generador Automático de la Lengua de Signos Española) de la Cátedra Indra-UCLM.

Abstract

According to the information of the CNSE (State Federation of Deaf Persons) there are more than 100.000 people in Spain with deep auditory deficiencies which use the Spanish Language of Signs (LSE) as way of communication. Nowadays, the first solutions are developing software of automatic translation to this language.

This MSc Final Project involves the construction of a motion capture system that takes real video sequences in the area of the LSE as input. In the videos, an interpreter uses LSE in an supervised environment (camera position) but without any active or passive elements that could facilitate the capture. These restrictions can reduce the recording times of the signs without any hardware specific motion capture hardware.

Thus the main goal of the project is the creation of a tool for analysis and edition of movement that may facilitate the construction of a dictionary of signs. This output will be in a standard format based on XML that will be able of being used in different virtual characters. Specifically, we propose its use in the area of GANAS project (Automatic Generator of Spanish Sign Language) of the Indra-UCLM research project consortium.

Dedicado a mis Padres y a mi Hermana, por su apoyo, preocupación y ayuda constante, y tratar de levantarme el ánimo cuando lo necesitaba, y a Cristina, mi otra mitad, por pasar tardes y tardes enteras junto a mi para que todo me resultara más sencillo y poder seguir adelante, y por su imaginación y ayuda para el bautizo de éste, el que ha sido "nuestro hijo" durante este periodo de tiempo. Os quiero.

Agradecimientos

Quiero mostrar mi más sincero agradecimiento a todas aquellas personas que de manera directa o indirecta han contribuido al desarrollo de este trabajo, en especial a mi tutor y amigo Carlos González Morcillo, por su ayuda, ideas y completa disponibilidad para atenderme y guiarme en la elaboración de este sistema.
A los miembros del grupo ORETO por cederme un lugar en su laboratorio cuando lo he necesitado, y por su apoyo y amistad.
A mis compañeros y profesores que me han acompañado durante la carrera.
Y a mi familia y amigos, que sin ellos esto no tiene demasiado sentido.

Gracias.

Índice general

Índice de figuras

Índice de tablas

Lista de algoritmos

Capítulo 1

Introducción

Las personas que sufren discapacidades auditivas se encuentran con barreras que a las personas oyentes les resulta difícil de apreciar, como el acceso a determinada información que resulta imprescindible en el día a día, ya sea información de salidas y llegadas de vuelos, autobuses o trenes, visionado de películas, reportajes, descripción de la realización de trámites en entidades públicas y privadas, y acceso a nuevas tecnologías.

Según datos de la CNSE (Confederación Estatal de Personas Sordas) [3], en nuestro país hay casi un millón de personas con diferentes tipos y grados de sordera. Para este colectivo, el lenguaje de signos, lengua utilizada por las personas sordas, adquiere la misma importancia como es la audición para oyentes. Actualmente, casi el 10 % de las personas sordas ya lo emplean.

Se trata de una lengua completa y muy diferente respecto a la utilizada por personas oyentes. El orden de las palabras en la oración no es como en la lengua común (Sujeto + Verbo + Complementos), sino que se anteponen las palabras clave, como sustantivos y adjetivos, al verbo principal de la oración.

La lengua de signos es la lengua natural de las personas sordas, pero en cada punto de la geografía mundial puede ser diferente, e incluso entre diferentes regiones de un mismo país. Sin embargo, si se establece comunicación entre dos personas de distintas partes del mundo, podrían entenderse ya que gran parte de las señas son icónicas.

A pesar de que la lengua de signos es una lengua completa (posee unas características gramaticales, sintácticas y léxicas propias), no ha sido reconocida legalmente como tal en España hasta el año 2007: LEY 27/2007, de 23 de Octubre, por la que se reconocen las lenguas de signos españolas y se regulan los medios de apoyo a la comunicación oral de las personas sordas, con discapacidad auditiva y sordociegas. "En la presente Ley se establecen las medidas y garantías necesarias para que las personas sordas, con discapacidad auditiva y sordociegas puedan, libremente, hacer uso de las lenguas de signos españolas y/o de los medios de apoyo a la comunicación oral en todos las áreas públicas y privadas, con el fin de hacer efectivo el ejercicio de los derechos y libertades constitucionales, y de manera especial el libre desarrollo de la personalidad, la formación en el respeto de los derechos y libertades fundamentales, el derecho a la educación y la plena participación en la vida política, económica, social y cultural".

Por ese motivo, sería deseable contar con herramientas que faciliten la comprensión del lenguaje de signos. Una de estas herramientas podría ser un sistema automático para la traducción y representación del lenguaje de signos.

Con el desarrollo de *GANAS* [27] se pretende ayudar a entender la información del entorno a personas con discapacidades auditivas y cuya información no pueden captar en situaciones donde la comunicación es principalmente oral. La solución desarrollada, mediante el uso de un personaje 3D, permite convertir un texto escrito en Lengua de Signos Española (LSE) a través del ordenador. Tecnológicamente, *GANAS* está formado por tres módulos: Editor de Gestos, Módulo de Traducción y Módulo de Representación. Concretamente, el módulo Editor se encargaría de calcular la posición y rotación de cada uno de los huesos del personaje 3D, a partir de vídeo real de una persona que signa, construyendo así un diccionario de gestos del cuerpo y movimientos faciales.

Actualmente, la edición para la creación de este diccionario de gestos se realiza empleando técnicas de animación basadas en cinemática inversa y *frames* clave. Pero esta edición es muy costosa en cuanto a tiempo se refiere; generar un signo puede llevar alrededor de dos horas para un animador experimentado.

Por ello, sería interesante desarrollar una herramienta que permita la captura de movimiento del lenguaje de signos, que permita ser integrado con el proyecto *GANAS* o con cualquier otro traductor automático a lengua de signos (el diccionario de signos es siempre necesario) y así llegar a dar una solución para eliminar esas barreras de comunicación que sufre la comunidad con discapacidad auditiva.

1.1. Justificación del trabajo

Con el propósito de reducir el tiempo empleado en la creación del diccionario de gestos para la representación de la lengua de signos mediante un actor virtual, siendo este diccionario imprescindible en traductores automáticos a lengua de signos y para la finalidad con la que ha sido creado el Proyecto *GANAS*, surge la idea de la creación de un *Sistema de captura semi-automática del movimiento para la representación de la Lengua de Signos Española (MOCASYM)*.

Se pretende que MOCASYM posea una interfaz amigable y lo más intuitiva posible, al alcance de usuarios no expertos en temas relacionados con la Visón por Computador o diseño 3D, que les permita crear nuevas palabras evitando la necesidad de conocer aplicaciones de animación 3D altamente complejas. Al ser una herramienta que debe detectar el movimiento de forma casi automática, facilitará, de una forma sencilla, la elaboración del diccionario de gestos para otras lenguas y adaptar más fácilmente el proyecto *GANAS* a otros países, simplemente, creando un nuevo diccionario de gestos a partir de vídeos que representen palabras en su propia lengua de signos.

Por otro lado, MOCASYM y su documentación asociada se distribuirá bajo licencia *GPL* [1], por lo que se utilizarán estándares y tecnologías distribuidas en términos de licencia de libre uso en busca de obtener la mayor portabilidad y escalabilidad posible.

1.2. Estructura del documento

El documento actual se estructura en seis capítulos y cuatro anexos. En el presente capítulo se presenta una introducción general a la problemática que surge, en cuanto a costes temporales, con la necesidad de creación de un diccionario de gestos para sistemas de traducción automática a lengua de signos (como el mencionado proyecto GANAS) cuya generación es muy costosa, justificando así el presente trabajo, cuyo diseño y desarrollo está basado en visión por computador sin marcas para facilitar su construcción. En el segundo capítulo se explicarán, de forma detallada, los objetivos del presente proyecto, así como el entorno de trabajo en el que se desarrollará. En el tercer capítulo se realizará un estudio del estado actual de las diferentes áreas que guardan relación con el proyecto así como algunas de las tecnologías existentes para su implementación. Este capítulo está dividido en los siguientes apartados:

1. **Visión por Computador**: Uno de los núcleos principales de la capa de negocio de la aplicación es el *tracking* sin marcas. En esta sección se estudian las principales técnicas de visión por computador empleadas en el proyecto.

2. **Técnicas de Representación 3D**: Para facilitar la interacción con el usuario del sistema se ha desarrollado un módulo de representación 3D que permite ver, evaluar y editar las trayectorias de los movimientos capturados de forma más real y cercana al usuario no experimentado.

3. **Toolkits para Interfaces Gráficas de Usuario**: La Interfaz Gráfica de Usuario (GUI) es la parte de cualquier sistema desarrollado que está en conexión directa con el usuario y que adquiere especial importancia para los usuarios poco experimentados.

[1] GNU Public License: http://www.gnu.org/licenses/gpl.html

4. **Lenguajes de marcas: XML** [2]: En la actualidad, son empleados para múltiples usos dada su flexibilidad, sencillez y estandarización.

En el primer apartado se hará una introducción a uno de los campos de la Inteligencia Artificial, como es la Visión por Computador, y se detallarán algunas bibliotecas existentes hoy en día para tratar temas relacionados con este campo y, más concretamente, las bibliotecas de OpenCV. En el segundo apartado se verán los fundamentos de la representación de gráficos 3D por computador; en concreto utilizando OpenGL como motor de renderizado. Como método de animación se estudiará también la clasificación de curvas de interpolación y aproximación, dando una visión general de ellas y profundizando el estudio en las Splines Cúbicas Naturales (utilizadas en el desarrollo del proyecto), estudiando las ecuaciones paramétricas que las definen y necesarias para su posterior implementación. En el tercer apartado se enumeran varias tecnologías libres y multiplataforma que permiten el desarrollo de una GUI. Se comentará con más detalle las bibliotecas GTK+ y glade, la elegida finalmente para este desarrollo, y algunas bibliotecas asociadas (como glib). Para terminar, se estudiarán brevemente los distintos lenguajes de marcado más utilizados y se comentarán algunas características importantes del lenguaje XML, que será el tipo de archivo de salida de la aplicación.

En el cuarto capítulo se abordarán los objetivos perseguidos, detallando la metodología utilizada en el desarrollo y la arquitectura del sistema, así como el ciclo de vida del software utilizado, módulos que componen el proyecto y algoritmos empleados más significativos. En el quinto capítulo se presentarán los resultados obtenidos con las distintas pruebas realizadas al sistema. Finalmente, en el sexto capítulo se exponen las conclusiones obtenidas del desarrollo del proyecto y las posibles mejoras, condicionadas sobre todo por las carencias que pueda tener el sistema y futuras ampliaciones y modificaciones para mejorar *MOCASYM*.

Al final del documento, se adjunta los diagramas de casos de uso y de secuencia empleados para el diseño y desarrollo del sistema, la guía de instalación de las bibliotecas necesarias para el funcionamiento del sistema, el manual de usuario y una breve descripción de los

[2] XML: siglas en inglés de E**x**tensible **M**arkup **L**anguage

módulos implementados para dar solución a los objetivos propuestos junto al código fuente de los más relevantes.

Capítulo 2

Objetivos del proyecto

Para satisfacer las necesidades citadas anteriormente, es imprescindible la creación de un diccionario de gestos reutilizable e independiente del actor virtual, del software de representación, y de las tecnologías específicas empleadas en el desarrollo del traductor. La construcción de un diccionario de gestos manualmente es una labor compleja que requiere gran cantidad de tiempo, incluso cuando se cuenta con herramientas de animación avanzadas, como descripciones de cinemática inversa y restricciones en la articulaciones del esqueleto virtual. En promedio, un animador experimentado puede tardar alrededor de 2 horas en animar un signo.

El objetivo principal de MOCASYM es la creación de un conjunto de herramientas que permitan la captura del movimiento de una persona intérprete de la LSE en un entorno controlado.

De este objetivo general, se definen una serie de subobjetivos:

- Captura automática del movimiento 3D a partir de una única cámara sin necesidad de marcas activas o pasivas. El proyecto, empleando las restricciones del dominio de la aplicación, capturará de forma automática la mayor cantidad de información del movimiento que le sea posible a partir de vídeo real. La figura 2.1 muestra el esquema de funcionamiento general de MOCASYM con las principales etapas relacionadas.

- Construcción de una interfaz gráfica de usuario con herramientas de edición avanzada adaptadas al dominio de la aplicación, que permitan editar y reparar movimientos aso-

ciados a las capturas anteriores. Esta interfaz de usuario permitirá igualmente añadir información adicional, difícilmente identificable únicamente con un flujo de vídeo, sobre el signo realizado (como el posicionamiento de manos).

- Generación de ficheros de salida independientes del actor virtual, esqueleto o software de aplicación que haga uso de ellos. Este subobjetivo permitirá reutilizar los gestos definidos en MOCASYM en diferentes paquetes software de traducción.

- Construcción del sistema multiplataforma basado en estándares abiertos y tecnologías libres. Para conseguir una amplia difusión en la comunidad de usuarios y facilitar la construcción de signos, el sistema se desarrollará para que pueda ser portado en diferentes arquitecturas empleando licencias libres.

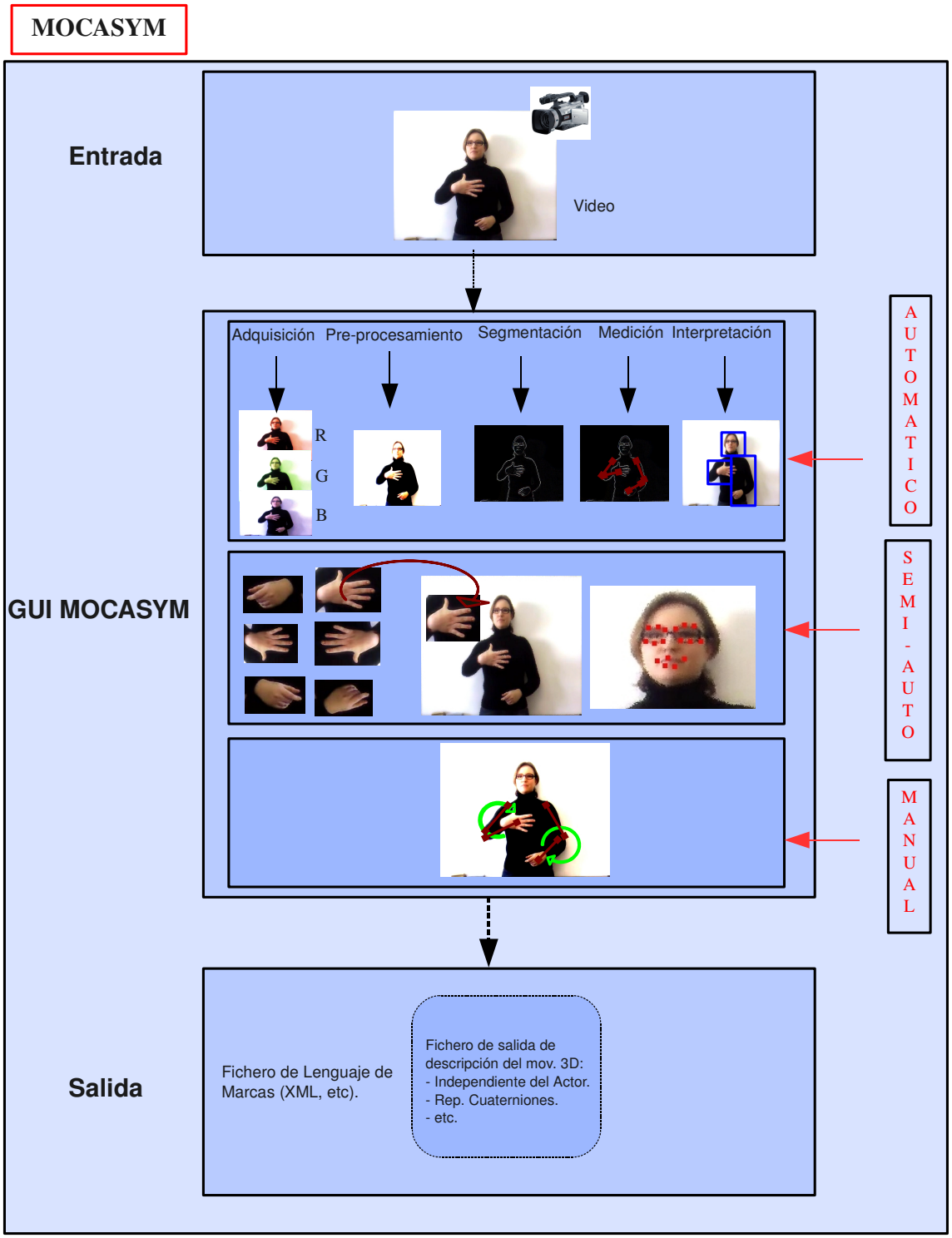


Figura 2.1: *Esquema General de los Objetivos del proyecto.*

Capítulo 3

Antecedentes, Estado de la Cuestión

3.1. Visión por Computador

Uno de los campos de la informática en los que se basará el proyecto a desarrollar es la Visión por Computador.

Las técnicas de visión por computador tienen como objetivo tomar decisiones útiles y eficaces sobre objetos del mundo real a partir del filtrado de una o varias imágenes. Es necesario construir una descripción del modelo para cada imagen. Su finalidad es la extracción de información del mundo físico a partir de imágenes utilizando para ello un computador. Gran parte del cerebro humano está dedicado a la visión y con la visión por computador se pretende llegar a límites tales como para ser capaz de comportarse de forma similar a la visión humana.

Sistemas de visión por computador

No es una tarea sencilla lograr que un computador emule el efecto de la visión humana mediante componentes electrónicos. El principal inconveniente surge porque el ojo humano percibe la luz reflejada por los objetos en un espacio en 3D, y los computadores intentan analizar estos objetos a partir de proyecciones en imágenes en 2D. El hecho de tratar con imágenes en 2D conlleva la pérdida de gran cantidad de información, produciendo un incremento de la dificultad del proceso de visión [44]. Las imágenes que debe analizar un computador son bidimensionales debido a que una escena del mundo real la representa mediante una rejilla rectangular. Cada uno de sus elementos se denominan píxeles. Cada píxel (*picture element*) puede almacenar diferentes valores dependiendo del formato de representación de la imagen.

Un sistema de visión por computador está compuesto por un sensor de imagen y un digitalizador. Un sensor de imagen es un dispositivo físico sensible a la energía electromagnética que genera una señal eléctrica en un instante de tiempo. La señal eléctrica suele ser una señal analógica. Es aquí cuando cobra importancia un digitalizador: dispositivo capaz de convertir la señal analógica de salida del sensor de imagen en una señal digital para que pueda ser procesada por un computador.

Etapas en un proceso de visión por computador

Un proceso de Visión por Computador se puede subdividir en varias etapas. Estas etapas se engloban en dos grupos.

A uno de ellos pertenecen las etapas que realizan procesos de bajo nivel y, al segundo grupo, las que realizan un procesamiento de la imagen de alto nivel. Los procesos de bajo nivel que ejecutan estas etapas son, por ejemplo, obtener las características más básicas de la imagen, como bordes, regiones y otros atributos simples. Las de alto nivel son las encargadas de recoger las características extraídas en el nivel inferior para construir una descripción de la escena.

Las etapas involucradas en este proceso son:

- **Adquisición de la imagen**: se captura una proyección de la luz reflejada por los objetos de la escena en 2D.
- **Preprocesamiento**: se realizan tareas como eliminación de ruido, realce de la imagen...
- **Segmentación**: se realizan tareas como detección de bordes y contornos para detectar diferentes elementos en la escena.
- **Extracción de características de la escena**: se obtiene una representación formal de los elementos segmentados en la etapa anterior.
- **Reconocimiento y localización de objetos**: mediante técnicas de triangulación, etc, se localiza al objeto en el espacio 3D.
- **Interpretación de la escena**: proceso final que describe la escena a partir de la información obtenida en las etapas previas.

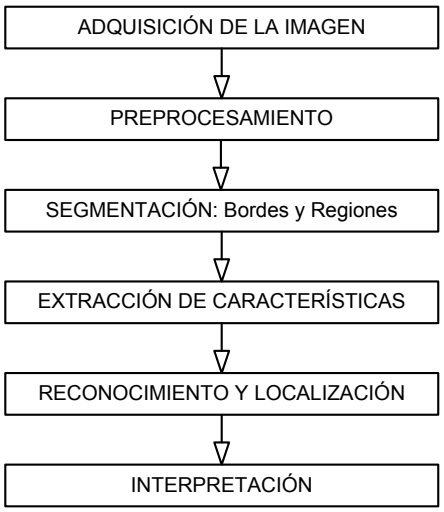


Figura 3.1: *Esquema de un proceso de análisis de imágenes: adquisición de la imagen, preprocesamiento, segmentación, extracción de características, reconocimiento y localización, e interpretación o clasificación.*

La Figura 3.1 muestra un diagrama de las etapas a considerar, generalmente, en un proceso de Visión por Computador.

3.1.1. Técnicas de captura de movimiento

La captura de movimiento consiste en grabar los movimientos realizados por un actor real y almacenarlos en un fichero de coordenadas 3D, para ser procesados por un computador con el fin de obtener una representación tridimensional de los movimientos capturados. Los puntos que se toman como referencia son las áreas del actor que mejor representan el movimiento de diferentes partes del mismo, como por ejemplo, en un ser humano se toman las articulaciones como puntos de referencia. Existen diferentes técnicas para capturar el movimiento, siendo las más extendidas los Sistemas Opticos, Magnéticos y Mecánicos.

3.1.1.1. Sistemas ópticos

Los **sistemas ópticos** utilizan los datos capturados de los sensores de la imagen para obtener la posición 3D de un objeto entre una o más cámaras calibradas proporcionando proyecciones solapadas. Estos sistemas producen datos con 3 grados de libertad para cada marcador (en el caso de la existencia de éstos como ayuda a la captura). Por ejemplo, la información de rotación del codo de una persona humana se podría obtener mediante marcadores en el hombro, en el codo y de la muñeca. Se pueden distinguir dos tecnologías: sistemas pasivos y sistemas activos.

- Sistemas pasivos: Las técnicas ópticas pasivas pueden ser aplicadas en un mayor número de situaciones que las técnicas ópticas activas. Sin embargo, al no utilizar una fuente de iluminación controlada el grado de incertidumbre con el que se realiza la correspondencia entre puntos de la escena y puntos de la imagen limitan, en muchas ocasiones, la exactitud de la medida.

- Sistemas activos: Las técnicas ópticas activas son consideradas las más interesantes desde el punto de vista de la reconstrucción tridimensional de una escena, dado que consiguen mayor precisión que las técnicas ópticas pasivas. Sin embargo, su implementación

resulta mucho más costosa debido, entre otros factores, a las estrictas condiciones de iluminación necesarias.

En los sistemas ópticos, generalmente, se utilizan entre 20 y 70 marcas y entre 2 y 6 videocámaras. Antes de comenzar la captura es necesario calibrar las cámaras y al iniciar la captura, las cámaras obtienen la posición (x,y) de cada marca produciendo un flujo en 2D. El siguiente paso es convertir este flujo 2D en posiciones 3D (x, y, z) mediante el uso de un software adecuado. Una de las ventajas que presenta este tipo de sistemas frente a los magnéticos es que ofrece al actor una mayor libertad de movimiento, pudiendo obtener animaciones más complejas. Por contra, este análisis requiere que las imágenes reúnan ciertas condiciones tanto de iluminación como de enfoque. Si estas condiciones no se cumplen, la precisión con la que se obtiene la información tridimensional de la escena se ve considerablemente reducida. A continuación [18] se describen las principales limitaciones que pueden aparecer al aplicar técnicas ópticas para la reconstrucción tridimensional de una escena.

Limitaciones de la técnicas ópticas

- *Iluminación adversa*. En los sistemas de visión por computador una iluminación incorrecta de la escena limita la obtención de información tridimensional de la misma. Si la escena está insuficientemente iluminada, a partir de la imagen capturada por el sensor será difícil extraer sus características. Si la escena está muy iluminada, el contraste en la imagen obtenida se reduce considerablemente, dificultando también la extracción de características de la escena. La solución a este problema suele pasar por ajustar el tiempo de exposición de la cámara para controlar la cantidad de luz que incide en el sensor de la misma. También se pueden utilizar filtros específicos que impidan el paso de ciertas longitudes de onda del espectro electromagnético al sensor y sólo permitan el paso de las necesarias para la reconstrucción de la escena, es decir, las longitudes de onda del patrón proyectado.

- *Oclusión*. La limitación más importante que presentan las técnicas ópticas es la oclusión, producida principalmente en los bordes de los objetos o en cambios bruscos de la geometría de su superficie. Esta limitación se presenta de dos formas diferentes:

La primera de ellas se denomina oclusión de cámara y se produce cuando, desde el punto de vista de la cámara, hay puntos de la escena que están ocultos por otros. La segunda recibe el nombre de oclusión de iluminación y se pone de manifiesto cuando existen ciertos puntos de la escena que producen sombras sobre otros, ocultando el patrón proyectado. Para evitar esta limitación, el diseño de la geometría del sistema de visión debe tener en cuenta las posibles oclusiones que se puedan producir y minimizarlas modificando la posición de la cámara y de las fuentes de iluminación. En caso de no poder eliminar todas las oclusiones con un diseño adecuado, se pueden utilizar varias cámaras y varias fuentes de iluminación para compensarlas.

- *Reflexión*. Si las superficies de los objetos son muy reflectantes, es posible que parte del patrón proyectado originalmente sobre la escena sea reflejado por la superficie de un objeto e ilumine otra parte de la escena (reflejo especular). La solución adoptada en los sistemas de visión para mitigar estos efectos es aplicar a la superficie de los objetos alguna capa de pintura que permita reducir su índice de reflexión.

- *Movimiento*. El movimiento de los objetos de la escena o del patrón proyectado sobre ésta puede ocasionar desenfoque en las imágenes adquiridas por la cámara del sistema de visión en el caso de que la frecuencia de muestreo del sensor sea relativamente baja en comparación con el movimiento de los objetos o del patrón. Este desenfoque provoca un efecto similar al del plegamiento de bordes durante el proceso de obtención de la información tridimensional de la escena a partir de las imágenes capturadas. Esta limitación está directamente relacionada con la iluminación adversa, ya que si se aumenta la frecuencia de muestreo del sensor para evitar el desenfoque, se reduce la cantidad de luz que incide en éste procedente de la escena.

- *Plegamiento de bordes*. En los bordes de los objetos parte de la luz del patrón proyectado sobre la escena se refleja y parte continúa su trayectoria recta al no entrar en contacto con la superficie del objeto. Este hecho motiva que la posición de un punto de la escena se calcule de forma errónea que, aplicado a todo el borde de un objeto, produce como resultado que dicho borde aparezca ligeramente plegado. El tamaño de este efecto es proporcional al ancho del patrón proyectado. La solución adoptada para

reducir este efecto es utilizar una fuente de iluminación que pueda ser focalizada de forma muy precisa, es decir, que pueda ser proyectada en áreas muy pequeñas, como por ejemplo un láser.

- *Coste elevado*. Algunos sistemas de captura de movimiento ópticos emplean sistemas de cámaras y entornos de trabajo muy sofisticados, lo que implica un alto coste en materiales. La solución para reducir costes se basa en el empleo de un menor número de cámaras o sistemas menos sofisticados produciendo un aumento en costes temporales en mejorar la calidad del software, ya que dificulta la captura de forma precisa y eficiente.

3.1.1.2. Sistemas magnéticos

Los **sistemas magnéticos** se basan en el uso de sensores con el fin de medir el campo magnético creado y se suelen utilizar en animaciones donde no son necesarios movimientos demasiados complejos. Los principales elementos de los que se compone son una serie de unidades de control electrónico y un ordenador conectado al sistema, que contiene el software necesario para poder comunicarse con los dispositivos. Una ventaja importante que presentan estos sistemas es que permiten obtener información en tiempo real, ya que permite comprobar si la animación es correcta, etc., y en caso contrario, realizar los cambios pertinentes. Pero por contra, el número de *frames* por segundo que captura es bajo (15 - 120 *fps*), tan sólo permite operar con un actor al mismo tiempo, tiene una gran sensibilidad al metal, no es recomendable para animaciones que requieren movimientos rápidos y el uso de cables limita en gran medida los movimientos que puede realizar el actor.

3.1.1.3. Sistemas mecánicos

Los **sistemas mecánicos** son aquellos que emplea un *hardware* adicional, como guantes y a otros elementos situados en el actor. Los movimientos son rastreados por hardware.

3.1.1.4. Optical Flow

Como alternativa de la técnicas ópticas sin empleo de marcas existe lo que se denomina *Optical Flow* o flujo óptico de imágenes.

El *Optical Flow* puede ser definido como el movimiento de los objetos respecto al observador, o lo que es lo mismo, es el movimiento relativo entre un observador y la escena visual.

Cuando la iluminación se mantiene a lo largo de una secuencia, el *Optical Flow* se corresponde con el campo de velocidades. Las aplicaciones del cálculo de velocidades son numerosas. Mediante el cálculo de la velocidad de diferentes objetos se pueden reconocer objetos en la escena (la forma de los mismos, su estructura externa, su orientación...).

El flujo óptico permite también el seguimiento de objetos en movimiento (*tracking* de objetos). Esta capacidad permite seguir un objeto físicamente, es decir, permite de alguna manera la captura de movimiento de un objeto en una escena visual.

Si la iluminación es constante, al moverse un punto de la imagen conserva su intensidad. Si se aplica este hecho a todos los puntos de la imagen, se deduce una ecuación que indique la componente de la velocidad, en dirección normal a los contornos. Para hallar la segunda componente de la velocidad deben hacerse algunas suposiciones.

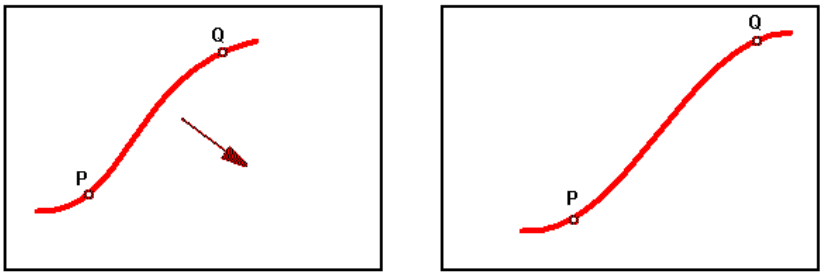


Figura 3.2: *Optical Flow. Desplazamiento de dos puntos de un objeto*

Con el hecho de obtener sólo información sobre la componente velocidad en la dirección del gradiente de intensidad (perpendicular a los contornos) surge un inconveniente denominado *problema de la apertura*, que es cuando no se puede determinar la componente del *Optical*

Flow en la dirección del contorno (ver figura 3.3).

Para solucionar el problema de la apertura se recurre a la técnica de criterios de vecindad, que consiste en mejorar regiones ambiguas mediante la observación de regiones próximas que presenten mejores características.

Es un problema muy complejo. Los distintos algoritmos basados en la Ecuación de Contención del *Optical Flow* se diferencian en los supuestos añadidos, que permiten obtener una segunda ecuación y, de esa forma, hallar la segunda componente de la velocidad.

En todo caso, el uso de estas técnicas diferenciales, implica la imposibilidad de hallar el *Optical Flow* en puntos pertenecientes a un contorno paralelo a la dirección del movimiento del objeto, ya que no nos ofrecen ninguna información referente a su velocidad, al ser nulo el movimiento en dirección perpendicular al contorno.

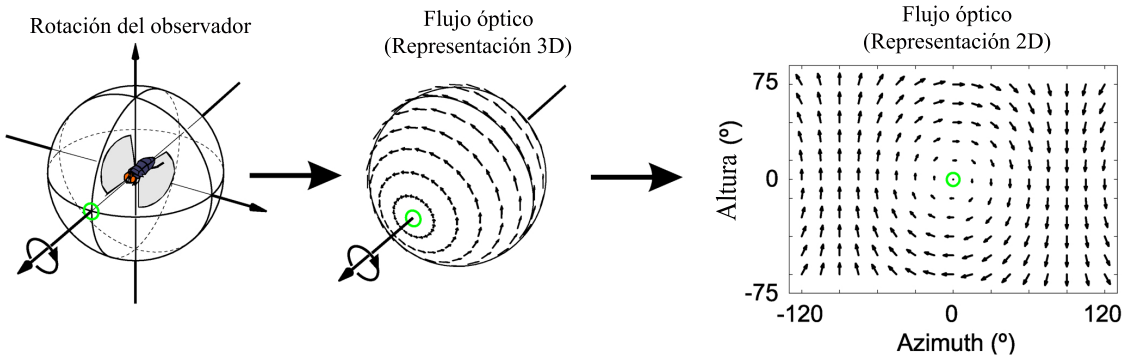


Figura 3.3: *Optical Flow experimentado por la rotación del observador. La dirección y magnitud de optical flow de cada punto se representa mediante la dirección y longitud de cada flecha (extraído de [11])*

Para dar solución a este problema, existen diferentes algoritmos. Algunos de ellos son:

- **Algoritmo de Horn-Schunck**: emplea una condición de suavidad como segunda condición. Para ello, se basa en el gradiente espacial y temporal. Para minimizárlos se emplea un factor de error.

- **Algoritmo de Lucas y Kanade**: permite extraer las dos componentes de la velocidad empleando una función para obtener los valores del gradiente espacial y temporal de

un entorno de vecindad del punto llevado a estudio, es decir, el problema de apertura es asumir que el movimiento no cambia en un determinado bloque de píxeles, denotado como $x \in B$, entre un *frame* y el anterior. Aunque este modelo no es adecuado para movimientos rotatorios, es posible estimar movimientos de traslación si el tamaño del bloque es suficientemente grande y cuenta con suficiente variación. Se define el error de la ecuación de flujo óptico sobre el bloque píxeles como:

$$E = \sum_{x \in B} \left(\frac{(\delta S_c(X,t)}{\delta x_1} v_1(t) + \frac{(\delta S_c(X,t)}{\delta x_2} v_2(t) + \frac{(\delta S_c(X,t)}{\delta t} \right)^2$$

- **Algoritmo de correlación de bloques**: trata de buscar en un entorno de vecindad de orden $7x7$ de un píxel determinado una ventana de orden $3x3$, permitiendo un mayor emparejamiento de la ventana a lo largo de la secuencia.

En términos de *Optical Flow*, cuando un objeto oculta a otro se dice que se ha producido una "discontinuidad". Estas situaciones deben ser reconocidas si se desea evitar que el algoritmo de flujo óptico empleado intente continuar la solución suavemente de una región a otra. Por tanto, un aspecto fundamental en el cálculo del *Optical Flow* será la segmentación. Combinando cálculo de velocidades y segmentación se puede mejorar de forma importante las estimaciones sobre velocidad y detección de movimiento.

El proceso de captura de movimiento de este proyecto está compuesto por un sistema óptico con una sola cámara, sin marcas, dificultando así la captura pero produciendo un ahorro, en cuanto a costes se refiere, muy elevado y empleando el algoritmo de Lucas-Kanade mejorado junto con segmentación de la imagen para la detección del movimiento.

3.1.2. OpenCV

OpenCV[1] es un conjunto de bibliotecas de funciones de programación cuyo uso principal es la Visión por Computador en tiempo real [36]. Su nombre proviene de los términos anglosajones *Open Source Computer Vision* [19]. Sus bibliotecas son de código abierto y desarrolladas en C/C++. Es multiplataforma y se puede ejecutar bajo Mac OS X, Windows,

[1]Sitio oficial de OpenCV: http://opencvlibrary.sourceforge.net/

GNU/Linux, etc. Está diseñado para que pueda ser emplado conjuntamente con la biblioteca de Procesamiento de Imágenes de Intel (IPL).

Algunos de recursos y operaciones que permiten las bibliotecas de OpenCV son:

- Procesado de imágenes y análisis de la misma.
- Capacidad de procesado desde diferentes fuentes de imagen o vídeo.
- Operaciones con vectores y matrices.
- Operaciones básicas y funciones de álgebra lineal.
- Estructuras de datos dinámicas como listas, colas y árboles.
- Procesamiento básico de imágenes entre los que se pueden destacar: filtros, detección de esquinas, conversiones de color, operaciones basadas en morfologías, histogramas y pirámide de imágenes.
- Análisis estructural, como análisis de contornos, transformación de distancias o aproximaciones poligonales.
- Calibración de cámaras y reconstrucción 3D.
- Reconocimiento de objetos.
- Etiquetado de imágenes.
- Análisis de movimiento.
- Interfaz gráfica.

A continuación se detalla las principales características por las que está compuesta esta biblioteca, como son los módulos por los que está formada, los tipos de datos más importantes y las funciones más representativas.

3.1.2.1 Módulos

Las bibliotecas de OpenCV se corresponden con 4 módulos bien diferenciados:

- Módulo *cv*: Contiene las funciones principales.
- Módulo *cvaux*: Funciones auxiliares de OpenCV, también contiene funciones que se encuentran en grado de experimentación.
- Módulo *cxcore*: Contiene las estructuras de datos y el soporte para funciones de álgebra lineal.
- Módulo *highgui*: Contiene las funciones GUI.

Es importante no confundir las funciones, con los tipos de datos propios de OpenCV. Para ello, la propia biblioteca utiliza una sintaxis distinta para cada caso, con ligeras diferencias, aunque en principio si no se presta la debida atención, es fácil confundir ambas sintaxis.

Cada una de las funciones referenciadas en OpenCV comienzan con las siglas "cv", seguida del nombre de la función, con la primera letra de cada una de las palabras que componen dicho nombre en mayúscula. Por ejemplo: *cvCreateImage*, *cvInvert*, *cvMatMulAdd*...

La sintaxis de los tipos de datos es muy similar a la de las funciones, aunque con la única diferencia de que los tipos comienzan con la siglas "Cv" y las funciones por "cv". Por ejemplo: *CvScalar*, *CvMat*... No obstante existen algunos tipos que se declaran de forma totalmente distinta (*IplImage*...). A continuación, se exponen los principales tipos de datos.

3.1.2.2 Tipos de datos en OpenCV

OpenCV proporciona tipos de datos básicos para su utilización. A continuación se describirán brevemente los más importantes:

- **IplImage**: Es el tipo de datos básico en OpenCV. Con este tipo de datos se representan todos los tipos de imágenes con sus componentes y características. Los campos de esta estructura ordenados son:

```
typedef struct _IplImage {
  int nSize; /* tamaño de la estructura iplImage */
  int ID; /* versión de la cabecera de la imagen */
  int nChannels;
  int alphaChannel;
  int depth; /* profundidad de la imagen en píxeles */
  char colorModel[4];
  char channelSeq[4];
  int dataOrder;
  int origin;
  int align; /* alineamiento de 4  o 8 bytes */
  int width;
  int height;
  struct _IplROI *roi; /* puntero a la ROI si existe */
  struct _IplImage *maskROI; /*puntero a la máscara ROI (si existe)*/
  void *imageId; /* uso de la aplicación */
  struct _IplTileInfo *tileInfo;
  int imageSize; /* tamaño útil en bytes */
  char *imageData; /*puntero a la imagen alineada */
  int widthStep; /*tamaño de alineamiento de linea en bytes */
  int BorderMode[4]; /*modo de borde (sup, inf, drch e izda)*/
  int BorderConst[4]; /*constes. para borde sup, inf, drch e izda.*/
  char *imageDataOrigin; /*puntero a la imagen sin alinear*/
} IplImage;
```

En la Tabla 3.1 se puede ver los componentes principales que forman esta estructura.

Es posible seleccionar algunas partes rectangulares de la imagen, lo que se conoce como regiones de interés (ROI). La estructura *IplImage* contiene el campo *roi*, que si no es nulo (*NULL*), apunta a la estructura *IplROI*, que contiene parámetros de la región seleccionada.

- **CvArr**: Es lo que se denomina un *metatype*, es decir, un tipo de dato ficticio que se utiliza de forma genérica a la hora de describir los parámetros de las funciones. *CvArr** se utiliza para indicar que la función acepta *arrays* de más de un tipo.

- **CvMat**: Estructura empleada para operar con imágenes. Es una matriz que se caracteriza porque aparte de almacenar los elementos como cualquier matriz, ofrece la posibilidad de acceder a información adicional que puede resultar de gran utilidad.

```
        typedef struct CvMat {
                int rows;       /* número de filas */
```

```
                int cols;        /* número de columnas */
                CvMatType type; /* tipo de matriz */
                int step;        /* no se utiliza */
                union {
                  float* fl; /*puntero a los datos de tipo float*/
                  double* db; /*puntero a datos de doble precisión*/
                }data;
        }CvMat
```

OpenCV: IplImage	
Componente	**Descripción**
widhtStep	número de bytes entre puntos de la misma columna y filas sucesivas
nChannels	indica el número de canales de color de la imagen
*imageData	puntero a la primera columna de los datos de la imagen
widht,height	anchura y altura de la imagen en píxeles
depth	información sobre el tipo de valor de los píxeles Los posibles valores del campo depth son los siguientes: - IPL_DEPTH_8U: Enteros sin signo de 8 bits (unsigned char) - IPL_DEPTH_8S: Enteros con signo de 8 bits (signed char o char) - IPL_DEPTH_16S: Enteros de 16 bits con signo (short int) - IPL_DEPTH_32S: Enteros con signo de 32 bits (int) - IPL_DEPTH_32F: Punto flotante con precisión simple de 32 bits (float)

Tabla 3.1: Descripción de los componentes principales de *IplImage*.

En todo programa implementado con OpenCV, este tipo de datos siempre irá asociado con la función *cvCreateMat* que permitirá configurar la estructura matricial de manera muy sencilla. Esta función se encarga de crear el encabezado de la imagen y de ubicar sus datos. Su estructura es:

```
CvMat* cvCreateMat(int rows, int cols, int type);
        rows: número de filas de la matriz
        cols: número de columnas de la matriz
        type: tipo de los elementos de las matrices.
        Se especifica de la forma:
                CV_<bit_depth>(S|U|F)C<number_of_channels>.
                Siendo:
```

```
                        bit_depth: profundidad de bit (8,16,31...)
                        number of channels: número de canales-matriz
                        (S|U|F): el tipo de datos de bit:
                                S: con signo
                                U: sin signo
                                F: flotante
```

- **CvScalar**: La estructura *CvScalar* es simplemente un vector de cuatro elementos. Ésta es muy útil a la hora de acceder a los píxeles de una imagen, sobre todo si es una imagen en color. La estructura *CvScalar* es la siguiente:

```
        CvScalar double
                val[4]; // vector 4D
```

- **CvPoint**: Define las coordenadas de un punto usando números enteros.

```
        typedef struct CvPoint {
                int x; /* coordenada x */
                int y; /* coordenada y */
        } CvPoint;
```

- **CvPoint2D32f**: Define las coordenadas de un punto usando punto flotante.

```
        typedef struct CvPoint2D32f {
                float x; /* coordenada x */
                float y; /* coordenada y */
        } CvPoint2D32f;
```

- **CvSize**: Estructura utilizada para definir las dimensiones de un rectángulo en píxeles.

```
        typedef struct CvSize {
           int width; /*anchura del rectángulo (valor en píxeles*/
           int height; /*altura del rectángulo (valor en píxeles)*/
        } CvSize;
```

3.1.2.3 Funciones más comunes en OpenCV

Algunas de las funciones más utilizadas y, por lo tanto, más importantes son:

- *void cvNamedWindow(char name, int type)*. Esta función crea una ventana gráfica. Sus parámetros son:

  - *name*: cadena de caracteres que sirve como nombre de la ventana.
  - *type*: formato de tamaño de la ventana: Se utilizará CV_WINDOW_AUTOSIZE, o se pondrá un 1 para seleccionar esta opción.

- *cvLoadImage(fileName, flag)*. Siendo:

  - *fileName*: nombre del fichero que se quiere cargar.
  - *flag*: características de carga en el fichero:
    $flag > 0$: se obliga que la imagen cargada sea una imagen de color de 3 canales.
    $flag = 0$: se obliga que la imagen cargada sea una imagen intensidad de 1 canal.
    $flag < 0$: la imagen se carga tal cual es, con el número de canales que posea su fichero.

  Cabe destacar que esta función puede recibir las imágenes en cualquier tipo de formato: BMP, DIB, JPEG, JPG, JPE, PNG, PBM, PGM, PPM, SR, RAS, TIFF, TIF, siempre y cuando los parámetros de la misma se adecuen a la imagen en cuestión.

- *void cvShowImage (char name, CvArr* img)*. Esta función dibuja la imagen indicada en la ventana correspondiente. Tiene como parámetros:

  - *name*: nombre de la ventana donde se dibujará la función.
  - *img*: imagen a dibujar.

- *void cvReleaseImage(CvArr* img)*. Esta función se encarga de liberar el espacio de memoria que ha sido asignado a una estructura *CvArr**. Posee un único parámetro:

  - *img*: es el nombre de la imagen que se desea liberar.

- *void cvDestroyWindows(char name)*. Esta función elimina la ventana que coincida con el nombre pasado como parámetro.

  - *name*: cadena de caracteres que indica el nombre de la ventana a cerrar.

- *void cvDestroyAllWindows()*. Esta función elimina todas las ventanas gráficas que hayan sido creadas previamente.

En el siguiente ejemplo se muestra un simple programa realizado con *OpenCV*, el cual carga una imagen de disco y la muestra en una ventana. En él se emplean algunas de las funciones expuestas anteriormente:

```
#include "highgui.h"

int main( int argc, char** argv ) {
        IplImage* img = cvLoadImage(argv[1]);
        cvNamedWindow("Example1", CV_WINDOW_AUTOSIZE);
        cvShowImage("Example1", img);
        cvWaitKey(0);
        cvReleaseImage(&img );
        cvDestroyWindow("Example1");
}
```

3.2. Técnicas de representación 3D

En esta sección se verán algunas de las técnicas de aproximación para realizar tareas de animación 3D por computador [45].

3.2.1. Métodos de animación basado en curvas de interpolación

El empleo de curvas es el denominador común de las técnicas básicas de animación 3D.

La característica principal de esta técnica es que se produce una interacción persona-computador para, en primer lugar, el animador establecer la posición clave de los objetos y, posteriormente, es tarea de la computadora de ir interpretando los valores de posicionamiento establecidos.

Generalmente, debido a la naturaleza del mundo real, existen fenómenos como la Gravitación Universal o el predominio de la unión unilateral de objetos que lo forman, los movimientos producidos por estos objetos se corresponden con trayectorias curvas. Por este motivo se

emplean métodos numéricos de interpolación como por ejemplo, mediante el uso de *splines* cúbicas naturales.

3.2.1.1 Representación paramétrica

Para la creación de curvas y superficies es muy habitual emplear ecuaciones matemáticas para definirlas. Éstas pueden estar en forma explícita, implícita y paramétrica.

La representación paramétrica facilita el tratamiento algorítmico para la representación de curvas [20]. Esta es una ventaja que hace que sea la representación paramétrica la más utilizada para la generación de gráficos por ordenador.

Si se establece u como parámetro, la ecuación de la curva será:

$$x = f_1(u)$$
$$y = f_2(u)$$
$$z = f_3(u)$$

En el ejemplo que sigue a continuación se muestra porqué la forma paramétricas es más ventajosa que la forma implícita.

Si se deseara dibujar un círculo cuyo centro está situado en el origen de coordenadas y con radio la unidad, en su forma implícita (la representación más común), se expresaría mediante una función de tipo cartesiana:

$$x^2 + y^2 = 1^2$$
$$z = 0$$

Sin embargo, su forma paramétrica sería:

$$x(u) = \operatorname{sen}(2\pi \times u)$$
$$y(u) = \cos(2\pi \times u)$$
$$0 < u < \pi$$

A la hora de dibujar el círculo, si se modifica u con valores que varíen entre 0 y 1 (en forma paramétrica, u está definida en el intervalo $[0, 1]$) se crearía una forma sencilla de conectar puntos que estén en el círculo, obteniendo como resultado final el contorno de la circunferencia. Dependerá del tamaño de los intervalos escogidos para que el resultado final tome una forma más o menos poligonal; si son lo suficientemente pequeños, aunque se tracen líneas rectas, la circunferencia quedaría perfecta.

Sin embargo, con la representación implícita el resultado no sería tan bueno. El cálculo computacional sería muy elevado debido a que habría que resolver dos raíces cuadradas para cada punto. Además surge un problema, y es que no podría implementarse debido a que las raíces cuadradas darían como resultado dos soluciones y sería inviable.

En general, este problema siempre se presenta para la representación de curvas mediante la forma implícita, lo que conlleva a que el trazado no sea posible.

3.2.1.2 Curvas cúbicas paramétricas. Curvas de Spline

La técnica de aproximación o interpolación es la técnica empleada como método de animación cuando la ecuación de la curva que describe el movimiento es demasiado compleja o, simplemente, no se encuentra. Para ello, la curva es evaluada como un conjunto de puntos, los cuales llevan asociada una descripción sobre su posición y rotación, en relación a unos ejes de coordenadas, en un determinado instante y se cambian en otro punto. Con esta técnica, con sólo almacenar los valores que han cambiado es suficiente para determinar la nueva posición sin necesidad de almacenar la imagen completa.

El principal problema que se plantea es la necesidad de editar la curva. Para la edición sería necesario modificar las funciones que describen la trayectoria de las mismas. Para hacer factible las modificaciones de las funciones es necesario fijar su forma. Por esta razón se utilizan funciones polinómicas, y se aproximan con curvas polinomiales a trozos.

Cada segmento de la curva global se indica mediante tres funciones (x, y, z), que son polinomios cúbicos en función de un parámetro. Las razones por las que se utilizan polinomios cúbicos son las siguientes:

- Las curvas de grado más pequeño que no son planas en tres dimensiones son polinómicas de grado tres.
- Si se utiliza una representación de menor grado la interpolación no será posible, es decir, que un segmento de curva pase por dos puntos extremos (calculados mediante la derivada en ese punto).
- Si se utilizan polinomios cúbicos con cuatro coeficientes, se emplean cuatro incógnitas para la resolución de estos. Los cuatro valores pueden ser los puntos extremos y las derivadas en ellos.
- Los polinomios de mayor grado requieren un mayor número de condiciones para el cálculo de los coeficientes y pueden formar ondulaciones que son difíciles de controlar.

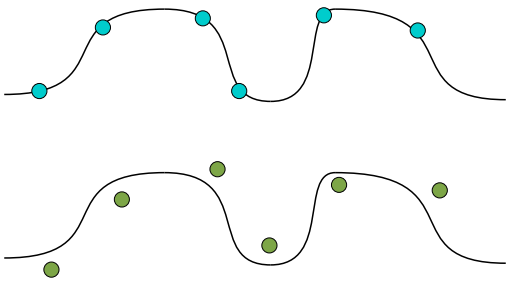


Figura 3.4: *Interpolación (figura superior) y aproximación (figura inferior).*

Los polinomios cúbicos [22] que definen un segmento de curva $Q(t) = [x(t)y(t)z(t)]$ tienen la forma:

$$x(t) = a_x t^3 + b_x t^2 + c_x t^1 + d_x$$

$$\begin{aligned} y(t) &= a_y t^3 + b_y t^2 + c_y t^1 + d_y \\ z(t) &= a_z t^3 + b_z t^2 + c_z t^1 + d_z \\ & 0 \leq t \leq 1 \end{aligned}$$

En las aplicaciones gráficas, las funciones de spline se caracterizan por estar constituidas por una serie de curvas que se unen entre sí cumpliendo algunas condiciones de continuidad en las fronteras de los intervalos.

Estas curvas son definidas mediante una serie de puntos llamados "puntos de control". Se distinguen dos casos:

- Interpolación. Un método interpola el conjunto de puntos de control cuando la curva pasa por ellos.
- Aproximación. Un método aproxima el conjunto de puntos de control cuando los polinomios se ajustan al trazado de los mismos, pero sin tener que pasar necesariamente por ellos (ver figura 3.4).

Un *spline* de grado k se define como [41]: *Dada una partición* Γ *de un intervalo* $[a, b]$*, una función spline* S *en* $[a, b]$*, de grado* $k \geq 0$*, correspondiente a la partición* Γ*, satisface:*
(i) S es un polinomio de grado $\leq$ *k en cada subintervalo* $(x_i, x_{i+1}), i = 0(1)(n-1)$
(ii) $S \in C^{k-1}[a, b]$

Un método clásico de interpolación es el polinomio de Lagrange. El término general de la fórmula de interpolación de Lagrange puede escribirse de la siguiente forma:

$$P(x) = \sum_{i=1}^{n} y_i \prod_{j=1}^{n} \frac{x - x_j}{x_i - x_j} \quad j \neq i$$

Un ejemplo de uso del mismo sería aproximar la función $y = \frac{12}{x^2+2x+5}$ mediante el polinomio de Lagrange y también mediante splines cúbicas naturales. En la Figura 3.5, se puede observar los resultados del experimento. En la zona superior están las gráficas resultantes de Lagrange y en la zona inferior las resultantes de splines. La gráfica punteada es la exacta a la función original.

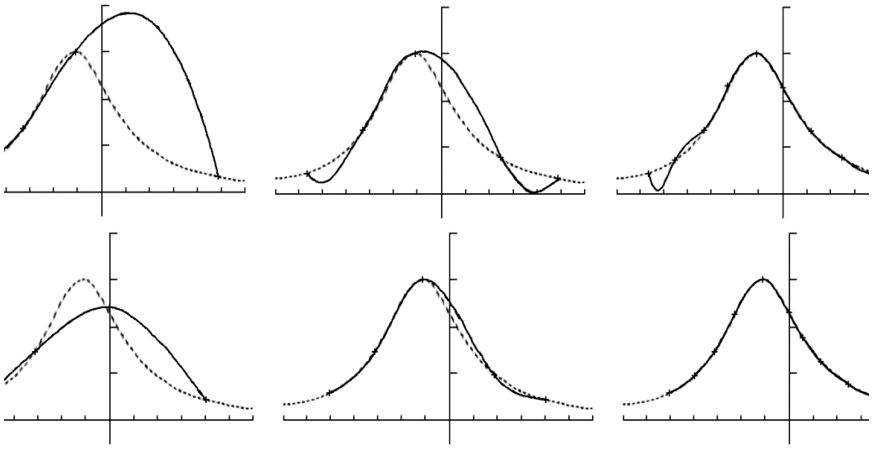


Figura 3.5: *Las gráficas de la parte superior corresponden (de izquierda a derecha) a la interpolaciónn de Lagrange con 4, 5 y 10 puntos. Las gráficas de la parte inferior corresponden a una interpolación con splines cúbicas naturales.*

Centrándose en el polinomio de Lagrange, con 4 puntos el polinomio conseguido es de tercer grado (la aproximación es mala y únicamente se limita a pasar por los puntos). Con cinco puntos, el polinomio es de cuarto grado (la aproximación sólo es buena entre el segundo y tercer punto, empezando desde la izquierda). Con 10 puntos, aparecen unas ondulaciones lejos del máximo de la función. Este fenómeno, se denomina "fenómeno de Runge", y aumenta su actividad cuando se incrementa el número de puntos. Usando *splines* cúbicas, la interpolación conseguida con 10 puntos es prácticamente perfecta.

Conclusiones:

- Hay que realizar muchas operaciones aritméticas en otros métodos de interpolación (como el de Lagrange), ya que habrá 2 bucles entrelazados, uno para el sumatorio y otro para el productorio.

- Si queremos añadir o suprimir un punto al conjunto de datos, habrá que volver a hacer todos los cálculos (este problema también lo presentan las splines naturales, aunque otras splines no lo tienen).

- En ciertos casos, un polinomio de grado alto puede desviar mucho la curva que pasa por los puntos dados (fenómeno de Runge).

3.2.1.3 Continuidad entre segmentos de la curva

Como se ha comentado anteriormente, las curvas están formadas por una serie de segmentos de curva unidos. Para evitar las posibles discontinuidades que se puedan provocar es necesario establecer unas condiciones de continuidad paramétrica para suavizar las transiciones entre segmentos de curva en el caso de ser necesario (ver figura 3.6).

Grado de continuidad se puede definir como el número de veces que se puede derivar la ecuación de una función continua:

- Continuidad de posición C^0 . La tangente a la curva en el punto es igual por la izquierda y por la derecha. Sean *P* y *Q* dos curvas, se dice que existe continuidad C^0 si $P(1) = Q(0)$.

- Continuidad de inclinación C^1 . La tangente es continua en todos los puntos de la recta, pero no en la curvatura. Sean *P* y *Q* dos curvas, se dice que existe continuidad C^1 si $P(1) = Q(0)$.

- Continuidad de curvatura e inclinación C^2 . La tangente y la curvatura es continua en todos los puntos. Sean *P* y *Q* dos curvas, se dice que existe continuidad C^2 si $P(1) = Q(0)$.

Para el estudio de las *Splines*, se detallará cada una de forma individual. Algunos tipos son:

- Splines cúbicas naturales.

- Splines de Hermite.

- Splines Cardinales y Catmull-Rom.

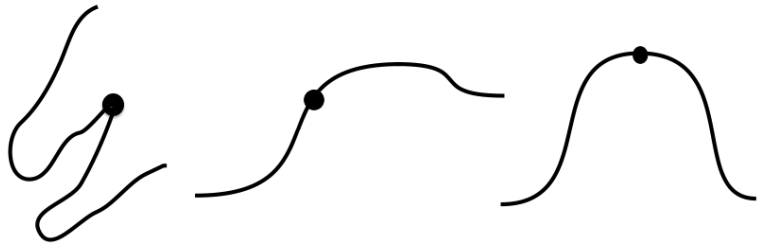


Figura 3.6: *Continuidad entre segmentos de curva. De izquierda a derecha: continuidad de orden 0, continuidad de orden 1, continuidad de orden 2.*

- Curvas de Bézier
- Curvas Beta-Splines.

3.2.1.4 Splines cúbicas naturales

Las Splines Cúbicas Naturales se caracterizan por poseer polinomios de grado 3 en cada uno de los intervalos.

Para n secciones de curva se necesitan $n+1$ puntos de control con $4n$ coeficientes que hay que calcular.

El grado de continuidad es C^2 , lo que significa que en la frontera entre dos secciones de curva adyacentes, la primera derivada y la segunda deben de ser iguales:

- Cada polinomio coincide con la función en ambos extremos (C^0).

$$p_i(u_i) = x_i; p_i(u_{i+1}) = x_{i+1}; i = 0, 1, ..., n-1$$

$$2n\ condiciones.$$

- La derivada primera (de los polinomios) es continua en los puntos de control (C^1).

$$p_i'(u_i + 1) = p_{i+1}'(u_{i+1}), i = 0, 1, ..., n-2$$

$$n-1\ condiciones.$$

- La derivada segunda (de los polinomios) es continua en los puntos de control (C^2).

$$p_i''(u_i + 1) = p_{i+1}''(u_{i+1}), i = 0, 1, ..., n - 2$$

$$n - 1 \; condiciones.$$

Considerando esto, se tienen $4n - 2$ condiciones.

El sistema debe ser compatible determinado, pero en este caso se trata de un sistema indeterminado al poseer $4n$ incógnitas y $4n - 2$ ecuaciones, por lo que es necesario obtener 2 ecuaciones más.

Hay varias alternativas para obtener el valor de los coeficientes. Una de ellas sería considerar que las segundas derivadas son nulas en los extremos u_0 y u_n.

El gran inconveniente que poseen estas curvas es el alto coste computacional que conlleva redibujar dichas curvas cuando se modifica cualquier punto de control. Además, no se tiene control local sobre ellas.

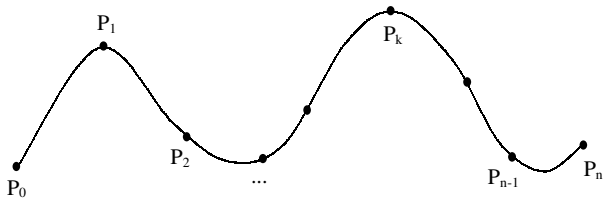


Figura 3.7: *Interpolación por piezas de spline cúbica (n+1 puntos de control).*

3.2.1.5 Splines de Hermite

Toman su nombre del matemático francés Charles Hermite. Su característica principal es que cada punto de control posee una tangente específica. Además, permiten control local.

La expresión general de las splines de Hermite es:

$$x(u) = x_k(2u^3 - 3u^2 + 1) + x_{k+1}(-2u^3 + 3u^2) + D_k(u^3 - 2u^2 + u) + D_{k+1}(u^3 - u^2) \quad (3.1)$$

Análogamente se definiría para y,z.

En la expresión anterior (Ec. 3.1):

- x_k es el valor de x en el punto P_k
- x_{k+1} es el valor de x en el punto P_{k+1}
- D_k es el valor de la primera derivada en P_k
- D_{k+1} es el valor de la primera derivada en P_{k+1}

Las splines de Hermite pueden ser útiles para algunas aplicaciones donde no sea difícil aproximar o dar valores a las pendientes de la curva. Sin embargo, generalmente es más útil generar valores para las pendientes de forma automática, sin requerir la entrada por parte del usuario. Estos cálculos de pendientes se suelen hacer en función de las posiciones de los puntos de control.

3.2.1.6 Splines Cardinales y Catmull-Rom

Los polinomios empleados son del tipo Hermite, pero se difieren en que las tangentes en las fronteras de cada sección de curva son calculadas en base a las coordenadas de los puntos de control adyacentes. Aparece el término de "parámetros de tensión", que son los que permiten ajustar más o menos la curva a los puntos de control. Para cada sección de curva se tienen las siguientes condiciones:

$$\begin{aligned} P_0 &= P_i \\ P_1 &= P_{i+1} \end{aligned}$$

$$P'_0 = \frac{1}{2}(1-t)(P_{i+1} - P_{i-1})$$
$$P'_1 = \frac{1}{2}(1-t)(P_{i+2} - P_i)$$

Como se puede observar, las pendientes en los puntos de control P_i y P_i+1 son proporcionales a las cuerdas formadas por P_{i-1} P_{i+1} , y P_i P_{i+2}.

Siendo t el parámetro de tensión, si se toma $t < 0$ la curva queda menos tensa alrededor de los puntos P_i P_{i+1} , y por el contrario quedará más tensa si se toma $t > 0$.

La expresión general de este tipo de curvas es:

$$\begin{aligned} P(u) = {} & P_{k+1}(-su^3 + 2su^2 - su) + P_k[(2-s)u^3 + (s-3)u^2 + 1] + \\ & P_{k+1}[(s-2)u^3 + (3-2s)u^2 + su] + P_{k+2}(su^3 - su^2) \\ & siendo\ s = \frac{(1-t)}{2} \end{aligned}$$

3.2.1.7 Curvas de Bézier

Fueron desarrolladas por Pierre Bézier quien, posteriormente, las utilizó para diseñar la diferentes partes de un coche (modelo Renault). Se basan en aproximación, siendo esta la razón principal por la que son más utilizadas en los sistemas CAD (*Computer Added Design*) que las splines cúbicas [20].

A medida que aumenta el número de puntos, también lo hace el grado de la curva. Para construir una curva de Bézier con continuidad C^2 es necesario $n = 3$, siendo la ecuación paramétrica:

$$P(t) = \sum_{i=0}^{n} P_i f_i(t),\ 0 \leq t \leq i$$

Las funciones $f(t)$ son polinomios de Bernstein de la forma:

$$f_i(t) = B_i^n(t) \begin{pmatrix} n \\ i \end{pmatrix} (1-t)^{n-i} t^i,\ 0 \leq t \leq 1 \tag{3.2}$$

Los puntos P (0), ..., P (1) que determinan una curva de Bézier, se denominan **puntos de control** y la poligonal que los une se llama **polígono de control de curva** (ver figura$_puntos_control$).

Veamos las propiedades de la curva:

- Si los puntos P_i están alineados, la curva será una recta. La curva es linealmente recta.
- Comienza en P_0 y termina en P_{n-1}.
- El vector tangente a la curva $P(t)$ en el punto P_0 , tiene la dirección del vector $\overrightarrow{P_0P_1}$.
- El vector tangente a la curva $P(t)$ en el punto P_n , tiene la dirección del vector $\overrightarrow{P_{n-1}P_n}$.
- La modificación de un punto de control afecta a toda la curva definida.

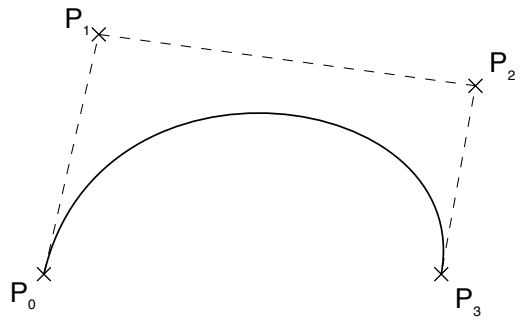


Figura 3.8: *Curva cúbica de Bézier donde se aprecian los puntos o nodos de anclaje $P_1 y P_2$.*

3.2.1.8 Curvas B-Splines

Las curvas de Bézier tienen algunas desventajas. Algunas de ellas son:

- El grado de la curva es dado por el número de puntos de control.
- No posee control local.

Para solventar este problema se pueden utilizar varios splines de Bézier e ir uniéndolos. El problema es que cuando el número de splines de Bézier utilizados es elevado, los cálculos se complican. Otra posible solución es emplear las B-splines que permiten definir de una vez todas las divisiones de curvas que formarán la curva global. La ecuación que define la B-Spline es:

$$P(t) = \sum_{i=0}^{n} N_{i,k}(t) P_i \tag{3.3}$$

Se determina la curva compuesta por varios tramos que son splines cúbicas. Para poder parametrizar de una sola vez la curva global, es necesario un intervalo de definición del parámetro. Las ecuaciones de cada intervalo están influenciadas tan sólo por *k* vértices del polígono de control, $2 \leq k \leq n+1$. Cada P_i son los $n+1$ vértices del polígono y $N_{i,k}(t)$ son las funciones mezcla:

$$N_i^k(t) = \frac{(t - x_i) N_{i,k-1}(t)}{x_{i+k-1} - x_k} + \frac{(x_{i+k} - t) N_{i+1,k-1}(t)}{x_{i+k} - x_{k+1}} \tag{3.4}$$

Al aumentar *k* aumenta también el grado de los polinomios, siendo *k* el orden de la curva, que cumple:

$$2 \leq k \leq n + 1 \tag{3.5}$$

La secuencia de valores de *x* se denominan nodos y definen una partición del intervalo en el que varía el parámetro. De esta forma se puede ajustar la región de influencia de cada punto de control. Al conjunto de nodos se le denomina **vector de nodos** y cumple:

$$x_k \leq x_{k+1}$$

$$0 \leq x \leq n + k$$

$$t_{min} = x_{k-1}$$

$$t_{max} = x_{k+1}$$

El vector de nodos permite delimitar la zona de influencia de cada punto de control relacionando la variable paramétrica *t* con los puntos de control P_i .

Se habla de B-Spline uniforme y periódico cuando todos los elementos del vector de nodos están equiespaciados. Esta configuración es la más utilizada en curvas cerradas, la curva

Curvas de interpolación				
	Grado polinomio	**Interpola**	**Carácter**	**Continuidad**
Lineal	1	Si	Local(2)	c^0
Bézier	n-1	No (sólo extremos)	Global (n)	C^∞
Spline local	3	Si	Local (4)	C^1
Spline global	3	Si	Global	G^2
B-spline	Ajustable	No	Local (ajust.)	Ajustable
B-spline (cúbico)	3	No	Local (4)	G^2

Tabla 3.2: Tabla comparativa de diferentes curvas.

no interpola a ninguno de los puntos de control y para crear una curva cerrada es necesario repetir los $k-1$ puntos desde el principio al final.

Se dice que un B-Spline es no periódico cuando los elementos del extremo del vector de nodos son iguales y los del centro se mantienen equiespaciados. En este caso, la curva sólo interpola a los puntos extremos y coincide con la curva de Bézier cuando el orden es máximo, $k = n + 1$. Se dice que el B-Spline es no uniforme cuando el vector de nodos no satisface ninguna de las condiciones anteriores.

3.2.1.9 Implementación de splines cúbicas naturales

Como ya hemos visto en otros apartados, la representación paramétrica de cada segmento que forman las splines cúbicas es del modo:

$$Y_i(u) = a_i + b_i u + c_i u^2 + d_i u^3 \tag{3.6}$$

De forma análoga se definiría la ecuación para X y Z (si fuera una curva en tres dimensiones). Como también se comentó, *u* varía desde 0 a 1 en cada segmento de la curva.

Como se ve en la figura 3.9, y acorde con la fórmula descrita anteriormente, el segmento i-ésimo va desde el punto de control V_i hasta el V_{i+1}. En la ecuación 3.6, $Y_i(u)$ representa $y(u)$ a lo largo del segmento i-ésimo. De forma similar podríamos definir $X_i(u)$ y $Z_i(u)$.

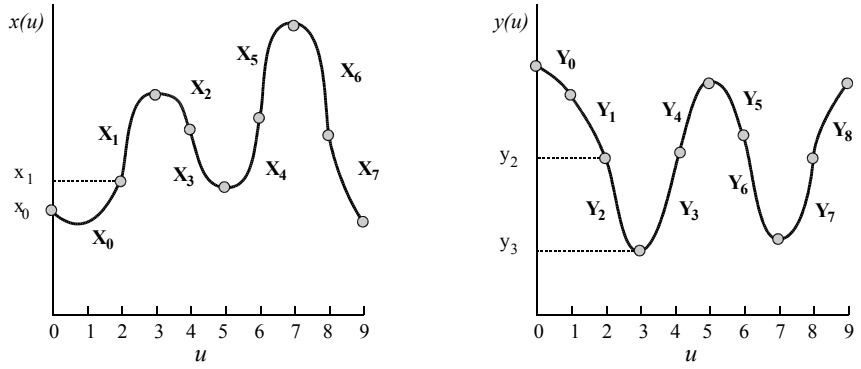


Figura 3.9: *Notación para nombrar segmentos en una spline cúbica de dos dimensiones.*

Como *u* varía desde 0 hasta 1 a lo largo de cada segmento de curva, entonces $Y_i(u)$ nos dará el punto de control i-ésimo (yi) cuando $u = 0$, e igualmente, el punto de control y_{i+1} cuando $u = 1$. Por tanto, $X_i(u) = xi$ cuando $u = 0$ y $X_i(u) = xi + 1$ cuando $u = 1$. Las cuatro incógnitas de la ecuación 3.6, que estarán presentes en cada segmento, se determinarán "emparejando" los puntos de control, forzando la continuidad de las primera y segunda derivadas en los puntos interiores y aplicando ciertas condiciones (que veremos más adelante) para los puntos extremos de la curva; primer y último punto.

A lo largo de cada segmento (tomando el segmento de curva i-ésimo), con el valor de los puntos extremos, se obtendrán las siguientes ecuaciones:

$$Y_i(0) = y_i = a_i \tag{3.7}$$

$$Y_i(1) = y_{i+1} = a_i + b_i + c_i + d_i \tag{3.8}$$

Recordar que el número de puntos de control es *n*, y el número de segmentos es *m (m = n-1)*. Las dos ecuaciones anteriores se obtendrán de todos los puntos de control interiores, por lo que se dispone de 2m ecuaciones para resolver las 4m incógnitas que se plantean inicialmente. Forzando a que las primeras y segundas derivadas sean iguales (respectivamente) en los extremos de cada intervalo, obtendremos otras 2m-2 ecuaciones. Por lo tanto, faltan 2 ecuaciones para resolver el sistema. Como se ha comentado, existen varias opciones para definirlas. Las splines cúbicas naturales toman su segunda derivada valor cero en los puntos inicial y final. Es decir: $Y_0''(0) = Y_{m-1}''(1) = 0$. Para resolver las incógnitas, resulta útil y

cómodo, introducir nuevas constantes D_i que representarán el valor de la primera derivada en el punto de control *i*. El valor de la primera derivada es:

$$Y_i'(u) = b_i + 2c_i u + 3d_i u^2$$

La primera derivada en los extremos del segmento *i*, nos da las siguientes ecuaciones:

$$Y_i'(0) = D_i = b_i \tag{3.9}$$

$$Y_i'(1) = D_{i+1} = b_i + 2c_i + 3d_i \tag{3.10}$$

La ecuación 3.9 y 3.10 nos darán realmente 2m ecuaciones (serán 2 para cada intervalo que definen los puntos interiores). Multiplicando la ecuación 3.8 por 3 y restándole 3.10 tendremos que:

$$\begin{aligned} 3Y_{i+1} &= 3a_i + 3b_i + 3c_i + 3d_i \; - \\ D_{i+1} &= b_i + 2c_i + 3d_i \\ \text{---------} & \text{----} \; \text{----------} \\ 3Y_{i+1} - D_{i+1} &= 3a_i + 2b_i + c_i \end{aligned}$$

Recordando las ecuaciones 3.7 que daba el valor de a_i, y la 3.9 el valor de b_i respecto de Di; sustituyendo:

$$c_i = 3y_{i+1} - 3y_i - 2D_i - d_{i+1}$$

$$c_i = 3(y_{i+1} - y_i) - D_{i+1} - 2D_i \tag{3.11}$$

Se obtiene el valor de c_i. De forma similar (multiplicando 3.8 por 2 y restando 3.10) se obtiene el valor de d_i:

$$d_i = 2(y_i - y_{i+1}) + D_i + D_{i+1} \tag{3.12}$$

De esta forma, se ha obtenido los valores de las incógnitas a, b, c, d para cada intervalo respecto de *y* y *D*. Las expresiones son:

$$a_i = y_i \qquad 3.7$$

$$b_i = D_i \qquad 3.9$$

$$c_i = 3(y_{i+1} - y_i) - D_{i+1} - 2D_i \qquad 3.11$$

$$d_i = 2(y_i - y_{i+1}) + D_i + D_{i+1} \qquad 3.12$$

Se calcula el valor de la segunda derivada en el punto i:

$$y_i''(u) = 2c_i + 6d_i u \tag{3.13}$$

Como se ha comentado, por las condiciones de spline cúbica natural, se tiene que cumplir que $y_i''(0) = y_{i-1}''(1)$, y se obtiene:

$$\begin{aligned} 2c_i &= 2c_{i-1} + 6c_{i-1} \\ c_i &= c_{i-1} + 3d_{i-1} \end{aligned}$$

Sustituyendo en la ecuación obtenida los valores de c_i y d_i (ec. 3.11 y ec. 3.12); sumando ambas y simplificando, se obtiene:

$$\begin{aligned} c_i &= 3(y_{i+1} - y_i) - D_{i+1} - 2D_i \; + \\ 3d_i &= 6(y_i - y_{i+1}) + 3D_i + 3D_{i+1} \\ - - - - - \quad & - - - \quad - - - - - - - - - - - - - - \\ c_i + 3d_i &= 3(y_{i+1} - y_i) + 6(y_i - y_{i+1}) + D_i + 2D_{i+1} \\ c_i + 3d_i &= 3(y_i - y_{i+1}) + D_i + 2D_{i+1} \\ \underbrace{c_{i-1} 3d_{i-1}}_{c_i} &= 3(y_{i-1} - y_i) + D_{i-1} + 2D_i \end{aligned}$$

Usamos de nuevo la ecuación 3.11, cambiando el valor de c_i por esa expresión:

$$\begin{aligned} 3(y_{i+1} - y_i) - 2D_i - D_{i+1} &= 3(y_{i-1} - y_i) + D_{i-1} + 2D_i \\ -D_{i-1} - 4D_i - D_{i+1} &= 3(y_{i+1} - y_i) - 3(y_{i-1} - y_i) \\ -D_{i-1} - 4D_i - D_{i+1} &= 3y_{i+1} - 3y_i - 3y_{i-1} - 3y_i \end{aligned}$$

Simplificando y sacando factor común:

$$\begin{aligned} D_{i-1} + 4D_i + D_{i+1} &= 3y_{i+1} - 3y_{i-1} \\ D_{i-1} + 4D_i + D_{i+1} &= 3(y_{i+1} - y_{i-1}) \end{aligned} \tag{3.14}$$

Por una de las condiciones de spline natural, se conoce que $Y_o''(0) = 2c_0 = 0$, por lo que, sustituyendo en la ec. 3.11 se obtiene que:

$$2D_0 + D_1 = 3(y_1 - y_0) \tag{3.15}$$

y como la condición también dice que:

$$Y''_{m-1}(1) = 2c_{m-1} + 6d_{m-1} = 0$$

Se obtiene (operando en ec. 3.11 y ec. 3.12; multiplicando la primera por 2 y la segunda por 6):

$$D_{m-1} + 2D_m = 3(y_m - y_{m-1}) \tag{3.16}$$

Las ecuaciones (3.14), (3.15), (3.16), que corresponden a las primeras derivadas Di, se pueden expresar en forma matricial de la siguiente manera:

$$\begin{pmatrix} 2 & 1 & & & & & & \\ 1 & 4 & 1 & & & & & \\ & 1 & 4 & 1 & & & & \\ & & 1 & ... & & & & \\ & & & 1 & 4 & 1 & & \\ & & & & 1 & 4 & 1 & \\ & & & & & 1 & 2 & \end{pmatrix} \begin{pmatrix} D_0 \\ D_1 \\ D_2 \\ ... \\ D_{m-2} \\ D_{m-1} \\ D_m \end{pmatrix} = \begin{pmatrix} 3(y_1 - y_0) \\ 3(y_2 - y_1) \\ 3(y_2 - y_2) \\ ... \\ 3(y_{m-2} - y_{m-3}) \\ 3(y_{m-1} - y_{m-2}) \\ 3(y_m - y_{m-1}) \end{pmatrix}$$

3.2.1.10 Optimización del método de resolución

La matriz $m \times m$ anterior es una matriz tridiagonal; los elementos que no están indicados implícitamente son ceros. Este tipo de matrices son muy cómodas para calcular el valor de la expresión por medio de un computador. Sin embargo, puede optimizarse todavía más si se logra hacer una de las 3 líneas diagonales obtenidas también cero. Consistirá en restar a cada fila la inmediatamente superior y multiplicarla por w_i. De esta forma, se transforma el sistema

anterior, obteniendo:

$$\begin{pmatrix} 1 & w_0 & & & & & & \\ & 1 & w_1 & & & & & \\ & & 1 & w_2 & & & & \\ & & & \dots & & & & \\ & & & & 1 & w_{m-2} & & \\ & & & & & 1 & w_{m-1} & \\ & & & & & & 1 & \end{pmatrix} \begin{pmatrix} D_0 \\ D_1 \\ D_2 \\ \dots \\ D_{m-2} \\ D_{m-1} \\ D_m \end{pmatrix} = \begin{pmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \\ \dots \\ \delta_{m-2} \\ \delta_{m-1} \\ \delta_m \end{pmatrix}$$

así, la matriz que contiene las ω_i únicamente tiene dos diagonales que no son ceros (acelerando los cálculos un 33 %). Los valores de ω y δ se calculan mediante las expresiones:

$$\begin{aligned} \omega_0 &= \frac{1}{2} \\ \omega_i &= \frac{1}{(4-\omega_{i-1})} && \forall i \in [1, m-1] \\ \omega_m &= \frac{1}{2-\omega_{m-1})} \\ \delta_0 &= 3(y_1 - y_0)\omega_0 \\ \delta_i &= [3(y_{i+1} - y_{i-1})\delta_{i-1}]\omega_i && \forall i \in [1, m-1] \\ \delta_m &= [3(y_m - y_{m-1})\delta_{m-1}]\omega_m \end{aligned}$$

Los casos particulares de cálculo son el primer y último punto de control. El resto, serán calculados de forma general por la expresión de término i.

Resolviendo la ecuación matricial, se aprecia que la derivada primera en el último punto de control se puede resolver directamente calculando el valor de δ. Será lo que se utilice en la implementación del algoritmo; primero se resolverá el término *m*, y volviendo atrás hacia el primero, se aplicará la ecuación general de término *i*. Se obtendrá:

$$\begin{aligned} D_m &= \delta_m \\ D_m &= \delta_i - \omega_i D_{i+1} && \forall i \in [m-1, 1] \end{aligned}$$

3.2.2. Representación 3D: OpenGL

La necesidad de implementar una herramienta que permita la de edición avanzada de la información tras el proceso de captura de movimiento que permitan editar y reparar movimientos detectados, surge la necesidad de realizar un estudio de la existencia de APIs que permitan trabajar con gráficos 3D.

A continuación, se estudiarán diferentes APIs de programación de alto nivel.

- **OpenGL:** OpenGL [17] es un estándar creado por Silicon Graphics en el año 1992, bajo el nombre de GL (Graphics Library), para el diseño de una biblioteca 2D/3D portable[2]. Es una biblioteca para la manipulación de gráficos 3D que, originalmente, fue escrita en C. Así, esta biblioteca puede usarse bajo multitud de sistemas operativos (Windows 95/98/NT/Xp/Vista, Linux, Unix, Solaris...) y lenguajes de programación (C/C++, Java, Visual Basic...). Su principal competidor es *Direct3D* de Microsoft Windows.

  La biblioteca se ejecuta independientemente de la capacidad gráfica de la máquina que se esté utilizando. Esto implica que la ejecución se dará por software si no se dispone de un hardware gráfico específico (como aceleradoras de vídeo, aceleradoras 3D, etc.).

- **Direct3D:** Es parte de DirectX [35]; API propiedad de Microsoft disponible tanto en los sistemas Win32 y Win64. Esta API fue diseñada para facilitar el manejo, trazado y transformaciones de entidades gráficas elementales, como líneas, polígonos y texturas, en cualquier aplicación que despliegue gráficos en 3D. Direct3D está provisto de una interfaz transparente con el hardware de aceleración gráfica. Su empleo es, generalmente, en aplicaciones donde el rendimiento de la máquina es fundamental, como los videojuegos, aprovechando el hardware de aceleración gráfica disponible en la tarjeta gráfica. Direct3D es uno de los múltiples componentes que contiene la API DirectX de Windows. Se le podría situar al nivel del GDI de Windows, presentando un nivel de abstracción entre una aplicación de gráficos 3D y los *drivers* de la tarjeta gráfica.

[2]Sitio oficial de OpenGL: http://www.opengl.org/

- **Mesa 3D:** Es una biblioteca gráfica de código abierto[3], desarrollada en 1993. Fue diseñada para renderizar gráficos tridimensionales en plataformas múltiples. Aunque Mesa no ha sido puesta en práctica oficialmente, ni licenciada por OpenGL, la estructura, la sintaxis y la semántica del API es la de OpenGL.

Tras el estudio realizado con algunas de la principales tecnologías para el desarrollo 3D, la elegida será OpenGL. Las características principales que han llevado a su elección es por cumplir uno de los objetivos definidos: Es libre y multiplataforma.

A continuación, se comenta con más detalle las características más importantes de OpenGL. Se divide en tres partes funcionales:

- La biblioteca OpenGL, que proporciona todo lo necesario para acceder a las funciones de dibujado de OpenGL.

- La biblioteca GLU (*OpenGL Utility Library*), una biblioteca de utilidades que proporciona acceso rápido a algunas de las funciones más comunes de OpenGL, a través de la ejecución de comandos de más bajo nivel, pertenecientes a la biblioteca OpenGL propiamente dicha [17].

- GLX (*OpenGL Extension to the X Window System*) proporciona un acceso a OpenGL para poder interactuar con un sistema de ventanas X Window, y está incluido en la propia implementación de OpenGL (su equivalente en Windows es la biblioteca WGL, externa a la implementación de OpenGL).

Además de estas tres bibliotecas, GLUT (*OpenGL Utility Toolkit*) proporciona una interfaz independiente de plataforma para crear aplicaciones de ventanas totalmente portables [30].

Los elementos físicos que configuran a un "pipeline" gráfico o a un sistema gráfico típico son los siguientes (ver figura 3.10)

- Entradas : todo aquello que se ha calculado y se desea dibujar... En definitiva, es el

[3] Sitio oficial de Mesa 3D: http://www.mesa3d.org/

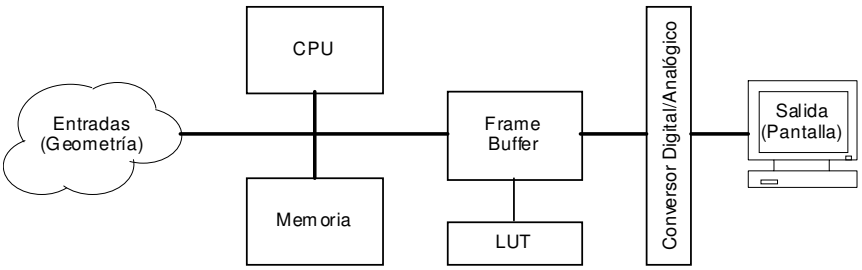


Figura 3.10: *Elementos físicos de un sistema gráfico*

"nuevo estado" tras algún evento que lo ha hecho cambiar, como por ejemplo, que la cámara se haya movido o alejado de la escena.

- Procesador ("CPU") : máximo administrador del sistema. Se encargará de gestionar la comunicación entre todos los módulos. Realizará operaciones según se le pida con ayuda de una/s ALU/s (Unidades Aritmético-Lógicas) y consultará la memoria cuando le sea necesario. En sistemas dedicados, y por tanto especializados en gráficos, podemos tener diversas CPU's trabajando en paralelo para asegurar un buen rendimiento en tiempo real (calcular y dibujar a la vez).

- Memoria : elemento indispensable y bastante obvio como el anterior. El "frame buffer" será la memoria que OpenGL utilizará para dibujar

- "Frame Buffer" : zona de memoria destinada a almacenar todo aquello que debe ser dibujado. Antes de presentar la información por pantalla, ésta se recibe en el frame buffer.

- Look Up Table (LUT) : contiene todos los colores disponibles en el sistema. Es conocida como *paleta de colores*. En sistemas "indexados" cada color tiene un identificador único.

- Conversor D/A : se encarga de transformar la información contenida en el frame buffer a nivel de bit (digital) a su equivalente analógico que puede ser procesado por un monitor CRT.

- Salidas : tras la conversión ya se dispone de información analógica para ser visualizada en pantalla.

3.2.2.1 Modelo conceptual de OpenGL

OpenGL utiliza un modelo de *cámara sintética* para interpretar una escena a dibujar. Básicamente se trata de imaginar un objeto situado en un determinado lugar y grabado por una cámara.

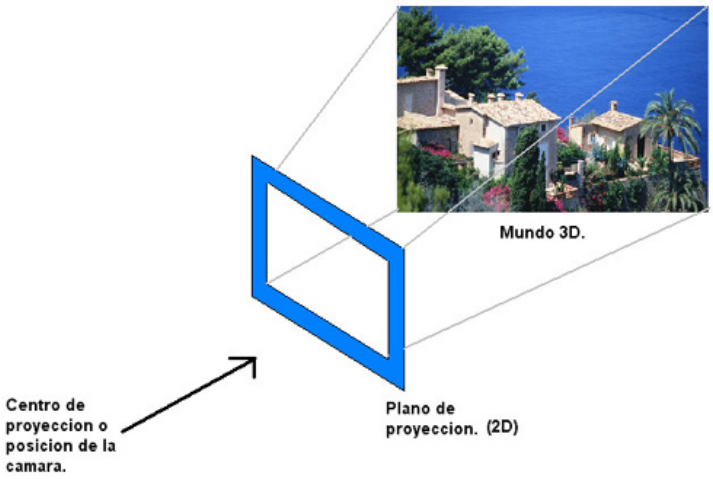


Figura 3.11: *Modelo conceptual de "cámara sintética" (extraído de [4])*

Como muestra el esquema de la Figura 3.11, se observa un mundo 3D desde una determinada posición (el centro de proyección). Aunque el mundo es tridimensional, su representación es plana (la pantalla del monitor): es el plano de proyección. Así, para pasar de coordenadas del mundo 3D a coordenadas del *frame buffer 2D*, es necesario proyectar. En este modelo de *cámara sintética* se distinguen elementos como:

- **Luces**: para iluminar el mundo 3D. Será necesario especificar la localización de cada uno de los focos de luz, intensidades y colores.

- **Cámara**: define en cada momento el punto de vista del mundo. Se caracteriza por su

posición, orientación y apertura (o campo visual). El campo visual es la cantidad del mundo 3D que la cámara alcanza ver.

- **Objetos**: Forman el mundo 3D y son filmados por la cámara. Se caracterizan por atributos de color, material, grado de transparencia... La cámara es independiente de los objetos ya que éstos se encuentran en una posición y la cámara en otra, por lo que se manejará por un lado la geometría de los objetos y por otro la posición de la cámara.

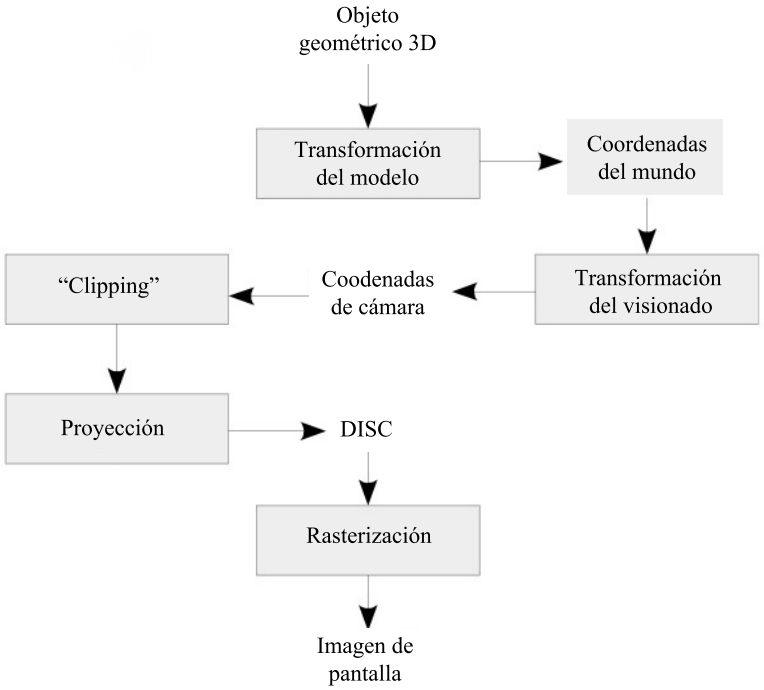


Figura 3.12: *Pipeline gráfico de OpenGL*

La forma de trabajar con OpenGL está condicionada por su arquitectura (ver Figura 3.12). El punto de entrada a la arquitectura de OpenGL es el objeto geométrico a dibujar, compuesto de líneas, puntos, polígonos... (primitivas). Inicialmente estos objetos tienen unos atributos que se podrán cambiar dinámicamente (se podrán trasladar, rotar, escalar, deformar, ...). Las operaciones de traslación, rotación y escalado se realizarán por el módulo de transformación del modelo. OpenGL realiza estas funciones multiplicando los elementos básicos de la geometría (vértices) por una serie de matrices, cada una de las cuales implementa un procedimiento (rotar, trasladar...). Tras haber transformado los vértices, se obtiene las posiciones de

los objetos en el mundo (coordenadas del mundo). Estas posiciones son relativas a un sistema de coordenadas que se define únicamente para el mundo creado (la cámara es independiente).

Con las coordenadas de la cámara calculadas y las posiciones de los objetos relativas a ella, es posible decidir qué objetos están dentro del campo visual de la cámara y cuales no. Este "recorte" lo realiza el módulo de *Clipping*, decidiendo qué información se representará en pantalla y cual no. Los objetos que deben dibujarse se proyectan mediante el módulo de proyección, pasando de las coordenadas 3D del mundo a coordenadas 2D del plano de proyección.

Tras proyectar se obtienen las coordenadas de pantalla independientes del dispositivo (D.I.S.C.[4]). En este punto, las coordenadas calculadas no se han asociado a ningún tipo de pantalla (resolución particular). El paso a píxeles físicos lo realiza el módulo de *rasterización*[5], que asocia los objetos dibujados en el DISC con coordenadas físicas que dibujará el CRT.

El pipeline gráfico se puede implementar vía software o hardware. En máquinas dedicadas como *Silicon Graphics*, todos los módulos están construidos en la placa base. Existen multitud de tarjetas aceleradoras 3D para PC que dan soporte para OpenGL, aunque no den soporte para la totalidad de los módulos. OpenGL da la misma salida independientemente de la implementación del pipeline. Así, aunque se cambie de máquina se obtendrá el mismo resultado (eso sí, a mayor o menor número de *frames* por segundo).

3.2.2.2 Funciones gráficas

OpenGL define sus propios tipos y convenciones de nombrado en las funciones. En vez de utilizar los tipos de datos estándar de C *int, float, double...* OpenGL define *Glint, Glfloat,*

[4]Del inglés *Device Independent Screen Coordinates*

[5]También nombrado como *scan conversion*

Gldouble, con prácticamente las mismas propiedades. De igual forma, las funciones comienzan con el prefijo "gl" (como *glVertex3f (0.0, 0.0, 0.0)*). El sufijo del nombre de la función indica el tipo de los parámetros que recibe (en este caso, 3 *Glfloat*). Si le pasáramos un vector de tres elementos *Glfloat*, la función a utilizar sería *glVertexfv (vector)*.

Podemos distinguir los siguientes tipos de funciones:

- **Funciones primitivas**: que definen todos los objetos a bajo nivel, como puntos, líneas y polígonos.
- **Funciones de atributos**: que permiten definir características de aquello que se dibujará.
- **Funciones de visualización**: utilizadas para posicionar la cámara, proyectar la geometría a la pantalla, recortar (clipping)...
- **Funciones de transformación**: Para girar, rotar, trasladar... la geometría.
- **Funciones de entrada**: Relativas al uso del teclado y del ratón, u otros dispositivos de Entrada/Salida.

La biblioteca OpenGL está dividida en tres módulos fundamentales. La parte del módulo **GL** será la más utilizada. Contiene todo el motor de renderizado y las primitivas básicas de la biblioteca. El módulo **GLU** (*Graphics Utility Library*) contiene instrucciones de más alto nivel para dibujar objetos comunes (esferas, cilindros...), controlar la cámara, asociar texturas a caras de objetos, etc... Por último, **GLUT** (*GL Utility Toolkit*) [30] incorpora además de funciones para dibujar objetos comunes, rutinas para controlar el programa desde el ratón y el teclado. Los tres módulos actúan directamente sobre el *frame buffer*.

OpenGL no puede dibujar polígonos cóncavos. Para dibujar este tipo de polígonos complejos, se utilizan una serie de triángulos. Todo polígono puede descomponerse en triángulos y, por tanto, a partir de éstos puede dibujarse cualquier forma. Esta técnica se conoce con el nombre de *Tesselation*. Ésta es la forma que utiliza OpenGL internamente para dibujar objetos complejos. Podemos recurrir a GLU para dibujar, por ejemplo, una esfera (llamando a *gluSphere*).

Los atributos se definen a nivel global, es decir, a partir de la llamada a una de estas funciones, la manera de dibujar ese atributo cambiará para todo el programa. Algunas de las más importantes son:

- **glClearColor** (0.0, 0.0, 0.0, 0.0): Indica el color con el que debe limpiarse cada frame cada vez que dibujemos la escena de nuevo. En este caso, el color de limpiado sería el negro. El cuarto parámetro de la función es el valor de "alpha", que indica el grado de transparencia (totalmente opaco en este caso).

- **glColor3f** (1.0, 0.0, 0.0): Indica el color de pintado mediante las componentes RGB. En este caso, indica que todo lo que se dibuje a partir de ahora será rojo (componente R con el valor máximo y G,B a cero).

- **glNormal3f** (1.0, 0.0, 0.0): Se asocia como vector normal de la cara de un polígono el definido desde el origen al eje positivo de las X.

- **GlMaterialfv** (GL_FRONT, GL_DIFFUSE, blanco): Cada objeto será de un material diferente de manera que los reflejos que se produzcan en él debido a las luces de la escena, variarán según su rugosidad, transparencia, capacidad de reflexión, etc... En este caso, se define que todas las caras visibles del polígono (FRONT), tendrán una componente difusa de color blanco (blanco es un vector de *glFloat* que define el color como 1.0, 1.0, 1.0).

3.2.2.3 Proyecciones

Una vez modelada la geometría 3D en OpenGL, debe dibujarse en una pantalla 2D dando sensación de profundidad. Para ello, hay que proyectar la geometría en un **plano de proyección**, que suele utilizarse por convención el *plano XY*. Un tipo de proyecciones muy utilizadas son las denominadas *planares*. En éstas se define una dirección de visión que va desde el observador hasta el objeto por medio de proyectores (líneas) que cortan el plano de proyección, generando así la imagen 2D que aparecerá en pantalla.

OpenGL maneja de forma nativa dos tipos de proyecciones planares: la ortográfica (subtipo de las planares paralelas) y la perspectiva de un punto. La proyección ortográfica cuenta con proyectores paralelos entre sí. El centro de proyección se encuentra en el infinito.

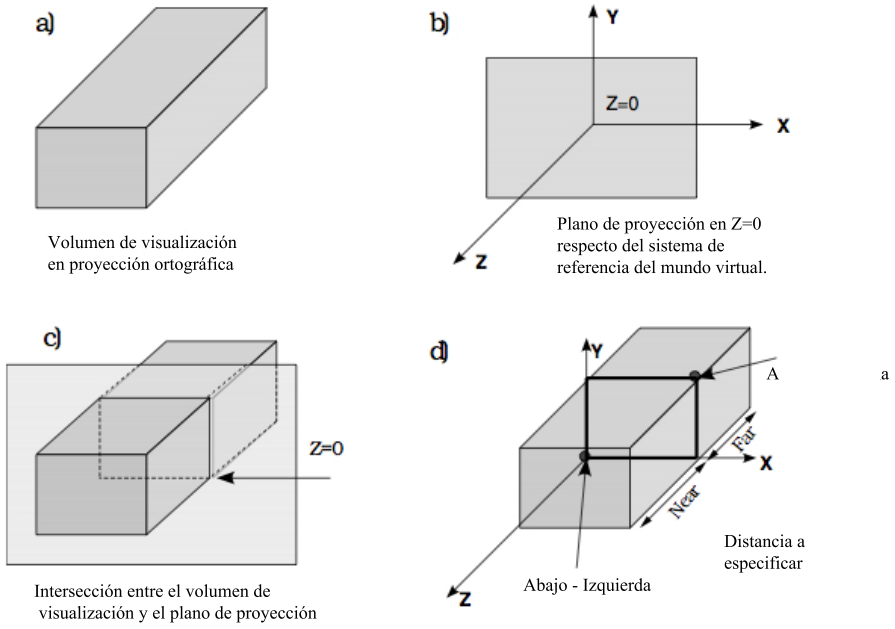


Figura 3.13: *Diagrama de cálculo de la proyección ortográfica de una escena*

La Fig 3.13, muestra un esquema simplificado de proyección ortográfica. Se trata de tomar todos los puntos (x, y, z) que se encuentren dentro del volumen de visualización (a), y eliminar su componente Z para poder dibujarlos en una pantalla 2D. De esta forma, todos los puntos pasan a ser del tipo $(x, y, 0)$ y se ha conseguido trasladarlos al plano de proyección (b y c). El resultado de la operación es como si se "aplastara" toda la geometría en el plano $Z = 0$, y este resultado se lleva a pantalla.

Es necesario especificar un volumen de visualización para indicar qué se desea renderizar. Este tipo de proyección tiene el inconveniente de que se pierde la noción de tamaño al acercar y alejar los objetos en el mundo, ya que todo se proyecta en $Z = 0$, por lo que el realismo conseguido no es total. Se utiliza tradicionalmente en herramientas de CAD/CAM.

Para utilizar este tipo de proyección en OpenGL, se llamará a la función glOrtho:

```
void glOrtho (Gldouble left, Gldouble right, Gldouble bottom,
              Gldouble top, Gldouble near, Gldouble far);
```

Nos permite especificar las distancias que se muestran en la Fig 3.13(d). Este modo de especificar el volumen de renderizado está relacionado directamente con el módulo de *Clipping*. Toda la geometría definida dentro del volumen saldrá en pantalla. Esto no quiere decir que no se haya calculado, precisamente para ver si se encuentra dentro o fuera del volumen de renderizado, hay que calcular las posiciones de los objetos en el mundo.

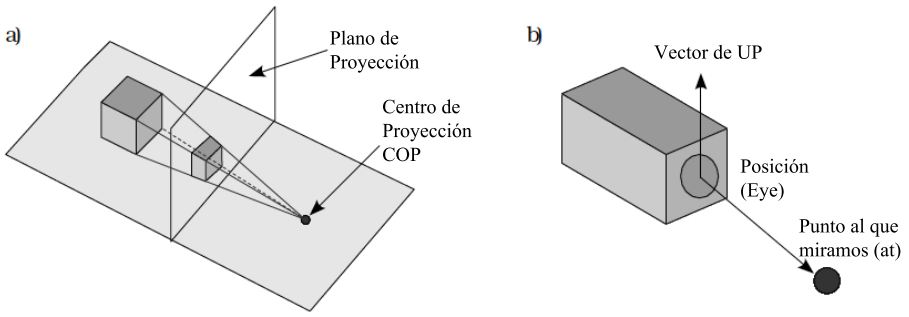


Figura 3.14: *a) Esquema de proyección perspectiva b) Parámetros de la cámara en OpenGL*

La proyección perspectiva se utiliza para dotar de mayor realismo a la visualización 3D, ya que sí preserva las dimensiones reales de los objetos al acercarse y alejarse de ellos. Un esquema de proyección perspectiva puede verse en la Fig. 3.14a). En esta figura se muestra una proyección perspectiva con un único COP (Centro de Proyección). Todos los proyectores salen de él y se dirigen hasta el objeto intersectando el plano de proyección. Para utilizar este tipo de proyección en OpenGL se llamaría a las siguientes funciones:

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluPerspective (FOVenGrados, RelacionAspecto, zCerca, zLejos);
```

FOVenGrados es el "field of view" o campo visual. Se refiere al ángulo de abertura vertical. *RelacionAspecto* o "aspect ratio" es el cociente entre la anchura y la altura del plano de proyección deseado. Los valores *zCerca* y *zLejos* son los vistos para la proyección ortográfica

del volumen de visualización. Las distancias *zCerca* y *zLejos* son siempre positivas y medidas desde el COP hasta esos planos que son, obviamente, paralelos al plano $Z = 0$. Dado que la cámara apunta por defecto en la dirección negativa de Z, el plano cercano (*near*) estará realmente situado en $z = -zmin$ mientras que el lejano (*far*) estará el $z = -zmax$.

3.2.2.4 La cámara

Todos los objetos que vea la cámara y estén dentro del volumen de visualización, serán proyectados. Los parámetros a definir para la cámara son:

- **Posición XYZ**. Al igual que cualquier objeto, la cámara debe posicionarse.
- **Orientación**. Una vez situada la cámara, debe orientarse.
- **Dirección ("at")**. Define hacia dónde mira la cámara.

En OpenGL se definen estos parámetros con la llamada a *gluLookAt*:

```
gluLookAt (eyeX, eyeY, eyeZ, atX, atY, atZ, upX, upY, upZ);
```

Esta función determina dónde y cómo está la cámara dispuesta. La posición de la cámara no tiene nada que ver con el tipo de proyección definida. La matriz que se modifica al llamar a esta función no debe ser $GL\_PROJECTION$ sino $GL\_MODELVIEW$. OpenGL calculará todas las transformaciones que aplicará al mundo 3D para que, manteniendo la cámara en el origen de coordenadas y enfocada en la dirección negativa de las Z, de la sensación de que se está observando todo desde un cierto lugar, pero para ello hay que activar la matriz antes de llamar a *gluLookAt*, llamando a *glMatrixMode (GL_MODELVIEW)*. Los parámetros de la función son los siguientes (ver Fig 3.14 b)):

- **Coordenadas de "eye"**. La posición XYZ donde colocar la cámara.
- **Coordenadas de "at"**. Valor XYZ del punto al que mirará la cámara.

- **Coordenadas del vector "up"**. Con este vector regularemos la orientación de la cámara. Este vector deberá "mirar hacia arriba". Cambiando el vector "up", variamos la orientación de la cámara, aunque siga mirando hacia el mismo punto y situada en idéntica posición.

3.2.2.5 Matrices y transformaciones geométricas

Como se comentó en la sección del *modelo conceptual de OpenGL*, toda la geometría generada pasa a través del pipeline gráfico sufriendo sucesivas transformaciones hasta su disposición final en pantalla. Estas transformaciones son matrices que multiplican a los vértices (vectores) modificando sus características.

En OpenGL se definen dos matrices principales que multiplicarán a toda la geometría: la **matriz de proyección** ($GL\_PROJECTION$) y la **matriz de modelado y visionado** ($GL\_MODELVIEW$). Cuando se define el tipo de proyección a utilizar, se están actualizando los valores de la matriz de proyección. Cada vez que la cámara se rote, traslade, escale o cambie de posición, se estará actuando sobre la matriz de modelado y visionado. La matriz de modelado-visionado se encarga de convertir el mundo 3D a 2D (es decir, crea la proyección). El primer paso a realizar en cualquier aplicación OpenGL es inicializar ambas matrices. Para ello, se les asocia la matriz identidad (*glLoadIdentity()*). Cualquier vector multiplicado por una matriz identidad no varía.

Un punto en coordenadas cartesianas se define como: Punto P = (x,y,z), y representa una determinada localización en el espacio 3D. Un vector, entendido como una resta entre dos puntos se define como *Vector* $V = P - P = (x, y, z) - (x, y, z) = (a, b, c)$. Por otra parte, se modela la geometría del objeto para luego transformala: trasladándola a otra posición, rotándola respecto de un eje, escalándola para cambiar su tamaño... Estas son las llamadas transformaciones afines[6]

[6]También denominadas en algunos libros como transformaciones rígidas o lineales. Se llaman así porque conservan las líneas. Dado un segmento definido por dos vértices, si se transforman los vértices y se unen de

Dado que las operaciones se realizan utilizando matrices, se necesita realizar una pequeña modificación por dos motivos:

- Para que no alteren de igual forma a un vector y a un punto (su representación es idéntica: tres números), lo cual sería incorrecto.
- Para poder efectuar algunas transformaciones afines (como la traslación) que de otra forma sería imposible realizar con una simple multiplicación de matrices.

Por estas razones es necesario introducir la representación con coordenadas homogéneas. Para convertir un vector o un punto de su representación en coordenadas cartesianas a homogéneas, basta con introducir una nueva coordenada a las típicas XYZ. Se le añade la componente W de la siguiente forma:
Punto $P_1 = (x_1, y_1, z_1)$ *en cartesianas, pasa a ser* $P_1 = (x_1, y_1, z_1, w_1)$ *en homogéneas.*
Vector $v = (a, b, c)$ *en cartesianas, pasa a ser* $v = (a, b, c, w)$ *en homogéneas.*
La nueva componente valdrá 1 en caso de puntos y 0 en caso de vectores. La transformación inversa se realiza normalizando los valores por su componente. En el caso del punto, tendríamos: $P_1 = (x_1, y_1, z_1, w_1)$ en homogéneas pasa a $P_1 = (x_1/w_1, y_1/w_1, z_1/w_1)$ en cartesianas.

Gracias a la función de escalado, es posible aumentar o disminuir el tamaño de un objeto. Únicamente hay que multiplicar cada uno de sus vértices por la matriz a) de la Figura 3.15, uniéndolos después con segmentos tal y como estaban al principio.

La traslación es una transformación afín imposible de realizar en cartesianas si no se incluye una suma de matrices. Lo interesante es únicamente multiplicar ya que, la mayoría de los pipelines gráficos implementados en hardware están optimizados en recibir matrices para concatenarlas y multiplicarlas.

---

nuevo, se obtiene como resultado final la línea que los une transformada. De esta forma, sólo será necesario aplicar transformaciones a los vértices de la geometría.

$$a)\begin{bmatrix} S_X & 0 & 0 & 0 \\ 0 & S_Y & 0 & 0 \\ 0 & 0 & S_Z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad b)\begin{bmatrix} 1 & 0 & 0 & T_X \\ 0 & 1 & 0 & T_Y \\ 0 & 0 & 1 & T_Z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$c)\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -sen\phi & 0 \\ 0 & sen\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad d)\begin{bmatrix} \cos\phi & 0 & sen\phi & 0 \\ 0 & 1 & 0 & 0 \\ -sen\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad e)\begin{bmatrix} \cos\phi & -sen\phi & 0 & 0 \\ sen\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figura 3.15: *Transformaciones geométricas afines. a) Escalar, b) Trasladar, c) Rotar EjeX, d) Rotar EjeY, e) Rotar EjeZ*

La rotación debe realizarse alrededor de un eje de referencia. Según sea el eje tomado, tendremos que utilizar una matriz de transformación u otra (ver Fig. 3.15, (c), (d) y (e)). El ángulo de rotación se define como positivo si supone girar en dirección contraria a las agujas del reloj, al mirar el eje sobre el que se rota de fuera hacia dentro (mirar hacia el origen).

La transformación de deformación consiste en hacer que alguna de las componentes de un vértice varíe linealmente en función de otra. Según muestra la Figura 3.16, en a) variamos las componentes Y y Z en función de X, en b) la X y Z en función de Y y en c) X e Y en función de Z.

3.2.2.6 Concatenación de transformaciones

Cuando se requieren aplicar múltiples transformaciones a un determinado objeto geométrico, es necesario concatenar todas las matrices por las que sus vértices deben multiplicarse. Para cada transformación se crea una matriz, multiplicándolas todas. De esta forma, se obtiene una matriz resultante, la única que se aplicará a los vértices para que se vean afectados por todas las transformaciones.

$$a)\begin{bmatrix} x_t \\ y_t \\ z_t \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ S_{xy} & 1 & 0 & 0 \\ S_{xz} & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad b)\begin{bmatrix} x_t \\ y_t \\ z_t \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & S_{yx} & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & S_{yz} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$c)\begin{bmatrix} x_t \\ y_t \\ z_t \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & S_{zx} & 0 \\ 0 & 1 & S_{zy} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Figura 3.16: *Deformaciones (Shearing). a) En función de X, b) En función de Y, c) En función de Z.*

La multiplicación de matrices no es conmutativa, por lo que el orden en que se multipliquen las matrices afectará al resultado final. En Robótica se utilizan vectores de tipo fila (o columna), que se multiplican por la izquierda y las matrices se ordenan de izquierda a derecha: se premultiplica. En gráficos por computador, los puntos se toman como vectores en columna que multiplican a las matrices por la derecha: se postmultiplica.

- Concatenación de Transformaciones en Gráficos: $[P_f] = [T_4] \cdot [T_3] \cdot [T_2] \cdot [T_1] \cdot [P_i]$
- Concatenación de Transformaciones en Ingeniería: $[P_f] = [P_i] \cdot [T_1] \cdot [T_2] \cdot [T_3] \cdot [T_4]$

En las ecuaciones anteriores, P_f es el punto transformado final, y P_i es el punto inicial a transformar. T_1 es la primera transformación a aplicar y T_4 la última. La multiplicación de las matrices, en ambos casos, se realiza de la forma habitual (de izquierda a derecha).

OpenGL utiliza una matriz especial donde guarda la información sobre todas las matrices que se han ido acumulando: la CTM (*Current Transformation Matriz*). Cualquier vértice que pase por el pipeline, será multiplicado por esta matriz. La CMT a su vez se compone de dos matrices: la **matriz de proyección** y la **matriz modelo-vista**. Éstas se concatenan, y de su producto se crea la CMT.

La matriz de transformación de modelo-vista debe entenderse como una pila. Es posible salvar el estado de la pila en cualquier momento, para recuperarlo después. Esto se realiza llamando a las siguientes funciones:

- Salvar el estado actual de la matriz: *glPushMatrix()*;
- Recuperar el estado de la matriz: *glPopMatrix()*;

Esto es útil cuando se quieran aplicar algunas transformaciones a una parte de la geometría, y el resto no deba verse afectada por estos cambios.

3.3. Toolkits para Interfaces Gráficas de Usuario

Se podría definir Interfaz Gráfica de Usuario (GUI) como un conjunto de formas gráficas e imágenes, encargadas de dar forma visual a la aplicación, y métodos que posibilitan la interacción de los usuarios con sistema a través de ese conjunto de objetos gráficos. [7] Una GUI es lo que el usuario puede ver cuando ejecuta una aplicación en cualquier terminal electrónico, y mediante la cual, interactúa con el sistema operativo y los datos almacenados en el dispositivo.

Por este motivo, las GUIs cobran especial importancia, sobre todo para un usuario inexperto que, aunque la aplicación esté muy bien construida y posea un cálculo computacional alto, ese software para ese usuario no será de calidad.

Algunas de las tecnologías existentes, que sean multiplataforma y libres de uso, son:

- **wxWidgets:** *Framework* especializado en el desarrollo de interfaces de usuario, programadas en C++, de aplicaciones multiplataforma. Está publicada bajo licencia LGPL. Dispone de adaptaciones de biblioteca o *bindings* para se usada con leguajes de programación distintos a C++. Proporciona una interfaz gráfica basada en bibliotecas nativas del sistema operativo. De esta manera, la aperiencia (o *Look and feel*) que toma la interfaz en cada momento dependerá del sistema operativo bajo el cual se está ejecutando.

[7] Con objetos gráficos se refiere a botones, iconos, ventanas, fuentes, etc. los cuales representan funciones, acciones e información.

- **Qt:** *Framework* para desarrollo de GUI multiplataforma escrito en C++ de forma nativa, y por lo tanto, orientado a objetos, proporcionando buena calidad visual y rapidez. Dispone de métodos de acceso a bases de datos mediante SQL, así como uso de XML, gestión de hilos, soporte de red, etc. Un ejemplo de uso de ésta podría ser el escritorio KDE de GNU/Linux.

- **GTK+:** Biblioteca para el desarrollo de interfaces gráficas de usuario, escrita en código C estructurado y liberada bajo licencia LGPL. No posee, por ejemplo, bibliotecas de acceso a bases de datos o de manejo y configuración de redes, pero se integra perfectamente con OpenGL y, además permite trabajar junto a *Glade*, que viene integrada en el Entorno de Desarrollo empleado (Anjuta), permitiendo total separación entre la interfaz y su funcionalidad. Por estas y otras características que se comentan a continuación, GTK+ es la biblioteca empleada para el diseño de MOCASYM.

3.3.1. GTK y glade

En sus comienzos GTK+ fue desarrollada como un conjunto de herramientas para el Gimp. GTK significa Gimp ToolKit (conjunto de herramientas GIMP) y GIMP quiere decir *Graphical Image Manipulation* (Manipulación de Imágenes Gráficas). *GIMP Toolkit* es una biblioteca utilizada para desarrollar aplicaciones que tengan una interfaz gráfica de usuario (GUI, *Graphical User Interface*). Esta biblioteca se utiliza ampliamente para desarrollar aplicaciones GUI para Linux. GIMP fue desarrollado con la biblioteca GTK+ y proporciona un ejemplo de aplicación con interfaz gráfica desarrollada profesionalmente. GTK+ es una biblioteca orientada a objetos escrita en C, que puede utilizarse en aplicaciones escritas en diversos lenguajes. Entre la lista de lenguajes permitidos están C++, Perl, Python, TOM, Ada95, Pascal, Eiffel...

GTK+ está liberada bajo licencia *GNU LGPL*[8], que permite la concesión de licencias flexibles de aplicaciones. También incluye GDK, o *Gimp Drawing Kit*, que es una interfaz de programación de aplicaciones (*Aplication Programming Interface*, API) dependiente de la

[8]GNU Library General Public License

plataforma y que se sitúa por encima de la API gráfica nativa (Xlib, Win32), proporcionando una API gráfica. Tanto GTK+ como GDK utilizan ampliamente la biblioteca GLIB, la base de la infraestructura de *Gnome*, que proporciona funciones para gestionar muchos tipos de datos comunes, como listas, árboles y cadenas, así como funciones de asignación de memoria y rutinas de tratamiento de errores.

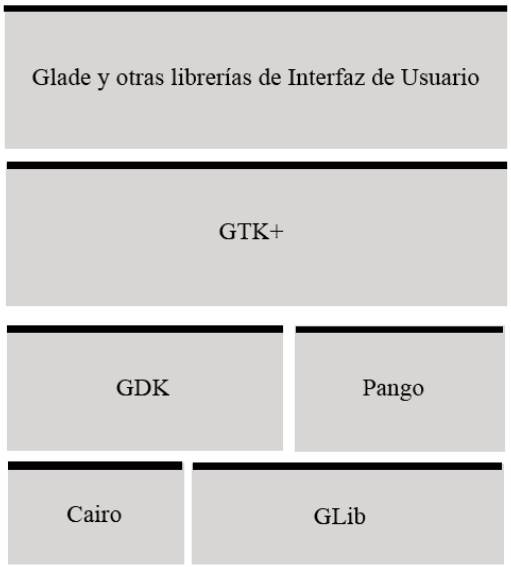


Figura 3.17: *Vista en capas de Bibliotecas gráficas y dependencias de GTK+.*

Al igual que la mayor parte de herramientas modernas para interfaces de usuario, GTK+ es una herramienta orientada a sucesos. La pantalla se construye mediante *widgets* (ventanas, etiquetas, botones, cuadros de texto...) y se establecen una serie de retrollamadas (*callbacks*) para esos *widgets*,con el fin de realizar el procesado basado en señales, que usualmente son sucesos de teclado o de ratón. Cuando una retrollamada recibe la notificación de una señal, la aplicación responde a dichas señales con algún tipo de procesamiento.

GTK+ es una biblioteca de *widgets* que utiliza GDK (GIMP Drawing Kit), que es un envoltorio alrededor de Xlib. GTK+ invoca a GDK para todo lo que se relacione con la visualización de los widgets.

Para construir una aplicación GTK+, cada archivo de la aplicación que utilice las funciones o definiciones de GTK+ debe incluir el archivo *gtk/gtk.h*, que es el archivo de inclusión principal de GTK+, donde se declaran todas las variables, funciones, estructuras, etc. que serán usadas en el programa. Además, es necesario enlazar la aplicación con un cierto número de bibliotecas.

3.3.1 Inicialización de la aplicación

El escribir un programa GTK+ requiere inicializar la biblioteca GTK+ con una llamada a la función *gtk_init*. Los argumentos de la aplicación *(argc, argv)* se pasan a la función *gtk_init* y se analizan en busca de alguna de las muchas opciones GTK+, que se emplean principalmente para la depuración.

```
/* Inicializar GTK */
gtk_init (&argc, &argv);
```

Esta función es la responsable de inicializar la biblioteca, para que pueda utilizarse, y de establecer algunos parámetros (como son los colores por defecto), establecer los controladores de las señales y comprobar los argumentos pasados a la aplicación desde la línea de comandos, buscando alguno de los siguientes:

```
--display
--debug-level
--no-xshm
--sync
--show-events
--no-show-events
--name
--class
```

Cuando encuentre alguno de estos argumentos, lo quita de la lista. De esta forma, en la lista sólo queda aquello que GTK no puede reconocer y quedando para ser tratado por el programa lo trate. Así se consigue crear un conjunto de argumentos que son comunes a todas las aplicaciones basadas en GTK. Después de inicializar GTK+, puede invocarse el resto de las funciones de la biblioteca GTK+.

3.3.2 Tipos de datos en GTK+

Muchos *widgets*[9] de GTK+ derivan, normalmente, de otros widgets. Por ejemplo, el widget de botón (*GtkButton*) se deriva del widget contenedor (*GtkContainer*), que a su vez se deriva del widget genérico (*GtkWidget*), que a su vez deriva de un objeto GTK+ (*GtkObject*). Todas las funciones de creación de widgets devuelven un puntero a un tipo de dato *GtkWidget*, que es un puntero a un widget genérico y puede necesitar ser convertido para determinadas funciones de widget. Un widget deriva de otro porque ese widget realiza la mayor parte de la funcionalidad de otro widget, y además, la funcionalidad propia que lo distingue de los demás.

Por ejemplo, la función de GTK+ de creación de un botón devuelve un puntero a un tipo de dato *GtkWidget*, no a un tipo de dato *GtkButton*. Esto permite que determinadas funciones genéricas, como *gtk_widget_show*, sean capaces de operar con todos los tipos de widgets. El dato *GtkWidget* devuelto por la función GTK+ de creación de un botón puede ser convertido a un dato *GtkButton* (utilizando la macro GTK_BUTTON) con vistas a emplearlo en funciones específicas de los botones. El mismo dato *GtkWidget* podría ser convertido en un dato *GtkContainer* (utilizando la macro GTK_CONTAINER) para poderlo utilizar con funciones de contenedor, porque el botón es un contenedor.

Aunque puede pasarse el widget de botón directamente a alguna de las funciones específicas de los botones, el compilador mostrará algún error si se pasa el botón como un widget genérico. La correcta programación en GTK+ requiere que se convierta el widget al tipo adecuado antes de invocar a una función de widget. Todo widget tiene una macro de conversión con la que puede convertirse un dato *GtkWidget* en cualquier tipo de widget GTK+.

Entre las macros de conversión de tipos, a continuación se detallan algunas de las más comunes. Estas macros comprueban si puede realizarse la conversión y, en caso afirmativo, la realizan.

[9] Cada uno de los componentes gráficos de GTK+

```
GTK_WIDGET(widget)
GTK_OBJECT(object)
GTK_SIGNAL_FUNC(function)
GTK_CONTAINER(container)
GTK_WINDOW(window)
GTK_BOX(box)
```

Todos los *widgets* derivan de la clase base *GtkObject*. Esto significa que siempre se puede usar un widget como argumento de una función, que acepte un objeto, realizando la conversión de tipo *GTK_OBJECT()*.

3.3.3 Funciones y retrollamadas

Las señales son necesarias en la programación de aplicaciones con interfaz gráfica de usuario, debido a que el programa deber ser capaz de responder a las acciones que el usuario realice. Si se mueve el ratón, se presiona un botón de la misma, se escribe un texto o se cierra una ventana, puede enviarse una señal a una función de retrollamada de la aplicación. Dicha señal puede ser una de las que la aplicación necesite gestionar.

GTK es un *toolkit* (conjunto de herramientas) gestionadas mediante eventos. Esto quiere decir que un programa GTK queda "dormido" en *gtk_main()* hasta que se recibe un evento, momento en el cual el control es transferido a la función encargada de tratarlo.

En una aplicación GTK+, se están generando señales continuamente, pero la mayoría de ellas son ignoradas. Por ejemplo, si se toma como ejemplo un botón, la aplicación tiene una serie de señales específicas del botón para cuando el usuario pulsa el botón del ratón, lo libera, hace clic con el ratón, lo mueve sobre el botón o cuando el ratón abandona el botón. Otros widgets son similares al widget del botón, y comparten que sólo una minoría de las señales son útiles para el desarrollado de aplicaciones, siendo la mayoría de las señales ignoradas.
Cuando es necesario tratar una señal, hay que asignar una retrollamada y asociarla con un widget. Los widgets pueden registrar retrollamadas y una misma retrollamada puede registrarse ante múltiples widgets.

Hay un conjunto de señales que todos los widgets heredan, como por ejemplo, *destroy* y hay señales que son específicas de cada widget, como por ejemplo, la señal *toggled* de un botón de selección.

Para que un determinado widget haga algo útil, habrá que crear el controlador que recoja la señal enviada por dicho widget, y se encargue de llamar a la función apropiada. Para hacer esto se usa la siguiente función:

```
gint gtk_signal_connect(GtkObject *object, gchar *name,
        GtkSignalFunc func, gpointer func_data);
```

donde el primer argumento es el widget que emite la señal, el segundo el nombre de la señal que se quiere capturar, el tercero es la función a la que se debe llamar cuando se captura la señal y el cuarto, los datos que queremos pasarle a dicha función.
La función especificada en el tercer argumento se denomina retrollamada (función de respuesta o *callback*) y debe tener la forma siguiente:

```
void callback_func(GtkWidget *widget, gpointer callback_data);
```

donde el primer argumento es un puntero al widget que emitió la señal, y el segundo un puntero a los datos que son pasados a la función. Estos datos son los que se le pasan a través del último argumento de la función *gtk_signal_connect()*. Otra llamada usada con el mismo propósito que la anterior es:

```
gint gtk_signal_connect_object(GtkObject *object,
                gchar *name,
                GtkSignalFunc func,
                GtkObject *slot_object);
```

Esta función es igual a la anterior, salvo en el último argumento, que en este caso es un puntero a *GtkObject*. En este caso la retrollamada deberá ser de la siguiente forma:

```
void callback_func (GtkObject *object);
```

3.3.4 Eventos

Además del mecanismo de señales, existe un conjunto de eventos, a los cuales pueden asociarse funciones capaces de responder cuando éstos se producen. Algunos de estos eventos son:

```
event
button_press_event
button_release_event
motion_notify_event
delete_event
destroy_event
expose_event
key_press_event
focus_in_event
focus_out_event
selection_notify_event
other_event
```

De todas ellas, las señales ***delete_event*** y ***destroy***, son de las más importantes. La primera de ellas se produce cuando una ventana va a ser destruida, mientras que la segunda se envía cuando la ventana esté siendo destruida. La ventana de nivel superior debe incluir una retrollamada para el suceso *delete_event*, porque este suceso indica que el usuario quiere cerrar la aplicación.

Para conectar una función de respuesta a alguno de los eventos anteriores se debe usar la función *gtk_signal_connect*, tal y como se comentó anteriormente, utilizando en el parámetro *name* uno de los nombres de los eventos que se acaban de mencionar. La función de respuesta para los eventos tiene un forma ligeramente diferente de la que tiene para las señales:

```
void callback_func(GtkWidget *widget,
                   GdkEvent *event,
                   gpointer callback_data );
```

GdkEvent es una estructura unión cuyo tipo depende de cual de los eventos anteriores haya ocurrido. Para que podamos decir qué evento se ha lanzado, cada una de las posibles alternativas posee un parámetro *type* que refleja cual es el evento en cuestión. Los otros com-

ponentes de la estructura dependerán del tipo de evento. Algunos valores posibles son:

```
GDK_NOTHING
GDK_DELETE
GDK_DESTROY
GDK_EXPOSE
GDK_MOTION_NOTIFY
GDK_BUTTON_PRESS
GDK_KEY_PRESS
GDK_KEY_RELEASE
```

Por lo tanto, para conectar una función de respuesta a uno de estos eventos se usará algo como:

```
gtk_signal_connect(GTK_OBJECT(button),"button_press_event",
        GTK_SIGNAL_FUNC(button_press_callback), NULL);
```

Aquí se asume que *button* es un widget *GtkButton*. Cada vez que el puntero del ratón se encuentre sobre el botón y éste sea presionado, se llamará a la función *button_press_callback*. Esta función puede declararse así:

```
static gint button_press_event (GtkWidget *widget,
        GdkEventButton *event,
        gpointer data);
```

3.3.5 Creación de ventanas

Un widget de GTK+ es un componente de interfaz gráfica de usuario. Ejemplos de widgets son las ventanas, casillas de verificación, botones y campos editables. Los widgets y las ventanas se definen siempre como punteros a una estructura GtkWidget. Esta estructura es un tipo de datos genérico, utilizado por todos los widgets y ventanas de GTK+.

Una vez iniciada la biblioteca GTK+, la mayoría de las aplicaciones crean una ventana principal. En GTK+, la ventana principal se denomina *ventana de nivel superior*. Las ventanas de nivel superior no tienen ventana padre, porque no están contenidas dentro de ninguna otra ventana. En GTK+, los widgets tienen lo que se denomina una *relación padre/hijo*, en la que el widget padre es el contenedor y el widget hijo es el incluido dentro de dicho conte-

nedor. Las ventanas de nivel superior no tienen ventana padre, pero pueden ser ellas mismas padre de uno o más widgets.

La creación de un widget en GTK+ es un proceso que se compone de dos pasos: primero se crea el widget y luego se hace visible. Puede aplicarse esta técnica para crear una ventana de nivel superior. Para ello, se invoca la función *gtk_window_new* con el parámetro *GTK_WINDOW_TOPLEVEL*. La función *gtk_window_new* devuelve un puntero a un dato de tipo *GtkWidget*. La ventana creada no es inmediatamente visible después de la creación, debiendo utilizarse la función *gtk_widget_show* para hacerla visible. Esta función le comunica a GTK que hemos acabado de especificar los atributos del widget, por tanto, puede mostrarlo.

```
/* Crear una ventana de nivel superior en gtk */
/* La ventana aún no es visible */
ventana = gtk_window_new ( GTK_WINDOW_TOPLEVEL );
/* Una vez creada, se debe hacer visible la ventana */
gtk_widget_show (ventana);
```

3.3.6 Introducción de widgets en la apliación

Los pasos generales a la hora de crear un widget son:

1. Usar *gtk_*_new*. Una de las diferentes formas de crear un widget.

2. Conectar todas las señales y los eventos a los controladores apropiados.

3. Establecer los atributos del widget.

4. Empaquetar el widget en un contenedor usando las llamadas apropiadas, como *gtk_container_add()* o *gtk_box_pack_start()*.

5. Mostrar el widget usando *gtk_widget_show()*. También puede usarse la llamada *gtk_widget_hide* para hacer que desaparezca el widget.

El orden en el que se muestran los widgets no es importante, pero se recomienda mostrar la ventana en último lugar, para que aparezcan de una sola vez todos los elementos del interfaz.

El hijo de un widget no aparece hasta que la propia ventana (widget padre) sea mostrada mediante *gtk_widget_show()*.

3.3.7 Bucle de sucesos de GTK+

Después de inicializar GTK+ y de situar en pantalla las ventanas y widgets, la aplicación cede el control de la ejecución a GTK+, para que se puedan procesar los sucesos (movimientos del ratón, pulsaciones de tecla, etc.). La llamada a la función *gtk_main* no finaliza hasta que la aplicación haga una llamada a *gtk_main_quit*. Pero, si *gtk_main* no termina de ejecutarse, ¿cómo puede la aplicación hacer una llamada a *gtk_main_quit*? En este caso, antes de realizar la llamada a *gtk_main*, deben crearse y configurarse una serie de retrollamadas para GTK+, de modo que ciertas señales devuelvan la ejecución a la aplicación para su preprocesamiento.

3.3.7 La biblioteca GLIB

La biblioteca GLIB es una colección de funciones comunes, ampliamente utilizadas en GTK+. Las listas enlazadas, los árboles, los mecanismos de tratamiento de errores, la gestión de memoria y los cronómetros son sólo una parte del contenido de esta biblioteca. GTK+ necesita la biblioteca GLIB y se apoya en ella para cuestiones de portabilidad y funcionalidad. Puede utilizarse la biblioteca GLIB sin GTK+ para desarrollar aplicaciones con interfaces de usuario no gráficas.

En lugar de utilizar los tipos estándar del lenguaje de programación C, GLIB utiliza su propio “conjunto de tipos”. Este enfoque facilita la portabilidad hacia otras plataformas y permite que los tipos de datos cambien sin necesidad de reescribir la aplicación. GLIB utiliza muchos tipos que son ligeramente distintos de los tipos de datos estándar en C. Lo que sería un tipo de datos *char* en C es un tipo de datos *gchar* en GLIB. Algunas de las modificaciones de tipos de datos más comunes son *gchar*, *gshort*, *gpointer* (equivalente a *void**), *glong*, *gint*, *gboolean*, *gstring*...

El utilizar los tipos de GLIB en una aplicación GLIB-GTK+ asegura que la aplicación continúe funcionando cuando cambie la implementación del tipo de dato subyacente (por ejemplo, gboolean). El tipo de datos gboolean podría ser definido, por ejemplo, como un tipo *int* en una versión posterior. El utilizar el tipo de datos *gboolean*, en lugar de *char*, asegura que la aplicación continúe compilando de manera limpia.

La biblioteca GLIB puede gestionar muchas estructuras de datos de uso común, incluyendo listas enlazadas y árboles, y proporciona un conjunto estándar de funciones para mensajes. Estas funciones son utilizadas por las biblioteca GTK+ y proporcionan una serie estándar de rutinas, disponibles en todas las plataformas.

Entre las muchas facilidades que aporta la biblioteca GLIB, hay funciones de tratamiento de cadenas, listas simples y doblemente enlazadas, árboles, tablas *hash*, relojes, errores, etc. Todas ellas con gestión transparente de memoria para el programador, además de facilitar la portabilidad del código entre plataformas.

3.3.8 Glade

Glade es una herramienta que asiste al desarrollo visual de interfaces gráficas de usuario mediante GTK+ en un entorno GNOME. Está liberada bajo licencia GPL y es totalmente independiente del lenguaje de programación empleado.

Su principal característica es que un diseño *glade* no genera código fuente, sino un fichero con formato XML donde almacena los elementos que configuran la GUI. Esto permite desacoplar completamente el aspecto visual de una aplicación de la lógica de negocio.

Para interactuar entre la interfaz gráfica y la lógica de negocio, existe una biblioteca llamada *libglade* que, mediante la función *glade_xml_get_widget(GladeXML *, gchar *)* , permite obtener un objeto de la interfaz en tiempo de ejecución con sólo pasarle el nombre del componente como parámetro.

3.4. Lenguajes de marcas

Un Lenguaje de marcas o de marcado se define como una forma de codificar un documento donde, junto con el texto, se incorporan etiquetas, marcas o anotaciones con información adicional relativa a la estructura del texto, su presentación,... Los lenguajes de marcas tienden a confundirse con los lenguajes de programación de uso habitual. Sin embargo, no son lo mismo, ya que el lenguaje de marcado no tiene entidades o funciones aritméticas ni variables, etc., como sí poseen los lenguajes de programación. El lenguaje de marcado se remonta a tiempos antiguos, habiendo sido empleado para la industria editorial y de la comunicación, así como entre autores, editores e impresores. Actualmente, se sigue empleando para ese fin.

Pueden hacerse dos clasificaciones sobre los tipos de lenguaje de marcas, en relación al contenido interno de los documentos y en relación al fin con el que han sido creados.
Una posible clasificación sería la que distingue entre procedimental, estructural e híbrido:

- **Procedimental**: enfocado hacia la realización de operaciones tipográficas.
- **Estructural**: enfocado a la descripción de la estructura lógica de un documento, pero no su tipografía.
- **Híbrido**: como su palabra indica, es una combinación de ambos.

Las hojas de estilo o lenguajes de transformación permiten la "traducción" de anotaciones de tipo estructural a anotaciones de carácter tipográfico.

Por otro lado, se puede distinguir entre lenguajes de marcado de presentación, de procedimientos y descriptivos o semánticos:

- **De presentación**: es aquel que indica el formato del texto. Este tipo de marcado es útil para maquetar la presentación de un documento para su lectura, pero resulta insuficiente para el procesamiento automático de la información. Este tipo de marcado resulta más fácil de elaborar, sobre todo para cantidades pequeñas de información. Sin embargo, es complicado de mantener o modificar. Esto ha provocado que su empleo

haya ido en decremento en proyectos grandes, siendo sustituido por tipos de marcado más estructurados. Se puede tratar de averiguar la estructura de un documento de esta clase buscando pistas en el texto. Por ejemplo, el título puede ir precedido de varios saltos de línea, y estar ubicado centrado en la página. Varios programas pueden deducir la estructura del texto basándose en esta clase de datos, aunque el resultado suele ser bastante imperfecto.

- **De procedimientos**: es aquel que está enfocado hacia la presentación del texto, sin embargo, también es visible para el editor del texto. El programa que representa el documento debe interpretar el código en el mismo orden en que aparece. Por ejemplo, para formatear un título debe haber, inmediatamente antes del texto en cuestión, una serie de directivas, e indicar al software instrucciones para centrar, cambiar a negrita, o aumentar el tamaño de la fuente. En sistemas más avanzados se utilizan macros o pilas que facilitan el trabajo. Algunos ejemplos de marcado de procedimientos son *nroff*, *troff*, TEX. Este tipo de marcado se ha usado extensivamente en aplicaciones de edición profesional, manipulados por tipógrafos calificados, ya que pueden llegar a ser extremadamente complejos.

- **Descriptivo o semántico**: en este tipo de marcado se emplean etiquetas para describir los fragmentos de texto sin especificar cómo deben ser representados o en qué orden. Algunos ejemplos de este tipo de lenguaje descriptivo son el SGML y el XML. Una de las virtudes del marcado descriptivo es su flexibilidad. El marcado descriptivo también simplifica la tarea de reformatear un texto, debido a que la información del formato está separada del propio contenido.
  El marcado descriptivo puede considerarse que ha evolucionado hacia el marcado genérico.

Debido al auge de aplicaciones en las que el usuario interacciona con un personaje virtual, se han desarrollado en los últimos años diferentes lenguajes de marcas con el fin de etiquetar las animaciones de los avatares. Dentro de las animaciones podemos distinguir la facial, la corporal, el diálogo y las emociones. Algunos de los lenguajes cubren todas ellas pero otros

como AML o CML tan sólo satisfacen a un subconjunto. A continuación, se describe el lenguaje XML, ya que la mayoría de los lenguajes orientados a la animación de avatares están basados en él. Por este motivo, el fichero de salida final que producirá MOCASYM estará basado en este lenguaje, para poder importarlo más fácilmente, mediante a algún procedimiento intermedio en algunos casos, en *suites* de producción 3D.

3.4.1. XML (eXtensible Markup Language)

XML (*eXtensible Markup Language*) no es, como su nombre podría sugerir, un lenguaje de marcado. XML es un meta-lenguaje que permite definir lenguajes de marcado adecuados a determinados usos. En nuestro caso, utilizaremos XML para definir un archivo de descripción de la escena en 3D, generado a partir de la captura del lenguaje de signos, a partir de vídeo, signado por una persona humana. Este archivo XML será la salida del sistema una vez finalizada la captura.

XML juega un papel importantísimo en la actualidad, que tiende a la globalización y la compatibilidad entre los sistemas. Esta tecnología permite compartir la información de una manera segura, fiable, fácil. Además, XML permite al programador y los soportes dedicar sus esfuerzos a las tareas importantes cuando trabaja con los datos, ya que algunas tareas tediosas como la validación de éstos o el recorrido de las estructuras corre a cargo del lenguaje y está especificado por el estándar, de modo que el programador no tiene que preocuparse por ello.

3.4.1.1 Documentos XML Bien Formados

Los documentos XML deben seguir una estructura estrictamente jerárquica respecto a las etiquetas que delimitan sus elementos. Una etiqueta debe estar correctamente incluida en otra (se escriben anidadas). Además, los elementos con otro contenido, deben estar correctamente cerrados. Por ejemplo, si se tiene:

```
<LI> Este es <B> un ejemplo <I> incorrecto </B></I>
```

```
<LI> En XML la <B> estructura <I> es </I> jerárquica </B>.</LI>
```

XML permite, al igual que HTML, elementos sin contenido indicándolo en su etiqueta, de la forma: <*elemento_sin_contenido/*>, por ejemplo: <*antecedente varnombre =“Vt” etiqling =“grande” /*>. Este elemento tendría dos atributos (*varnombre* y *etiqling*), pero no tendría contenido.

Los documentos XML sólo permiten un elemento raíz, del que todos los demás sean parte. Es decir, la jerarquía de elementos de un documento XML bien formado sólo puede tener un elemento inicial.

Los valores de los atributos en XML siempre deben estar encerrados entre comillas simples o dobles. Por ejemplo *¡sistema nombre=“Marcas Perdidas”¿* siendo nombre un atributo de sistema.

Al utilizar XML es necesario asignar nombres a las estructuras, tipos de elementos, entidades, etc... No se pueden crear nombres que comiencen con la cadena “xml”, “xMl”, “XML”, o cualquiera de sus variantes (mayúsculas y minúsculas). Se pueden incluir letras, guiones, números y caracteres de puntuación, pero el nombre sólo puede comenzar por letra. Las construcciones como etiquetas, referencias de entidad y declaraciones se denominan marcas y son la única parte del documento que el procesador XML puede entender.

Los documentos XML pueden (no es obligatorio) comenzar con unas líneas que describen la versión de XML, el tipo de documento y alguna información adicional. Por ejemplo:

```
<?xml version="1.0"?>
```

3.4.1.2 Elementos, Atributos, Secciones y Comentarios.

Los **elementos** de XML pueden tener contenido (más elementos, caracteres, o ambos a la vez), o bien ser vacíos. Un elemento con contenido siempre comienza con una ¡etiqueta¿que puede contener **atributos** o no, y termina con una ¡/etiqueta¿que debe tener el mismo nombre.

Al contrario que HTML, en XML siempre hay que cerrar un elemento.

Un elemento vacío, al no tener etiqueta de "cierre" que delimite un contenido, se cierra de la forma </etiqueta>.

Los elementos pueden tener atributos, que son una manera de incorporar características o propiedades a los elementos del documento. Siempre van encerrados entre comillas simples o dobles.

Los **comentarios** insertados en el documento XML serán ignorados en la etapa de procesado de la información. Tienen el mismo formato que los comentarios en HTML, es decir, van encerrados entre " <!−−" y "−− >".

Las **secciones** *CData* le indican al *parser* que ignore todos los caracteres de marcas que se encuentren en el interior de esta/s sección/es. Son muy útiles cuando queremos visualizar código XML como parte del texto. Todos los caracteres que existan entre el inicio (*<![CDATA[*) y el fin (*]]>*) son pasados directamente a la aplicación sin interpretación.

El único literal que no puede ser utilizado dentro de la sección es, lógicamente, el *]]>*. Ejemplo:

```
<![CDATA[ <!ENTITY amp "&"> <!-- &= ampersand -->
 <CODIGO>
     *p=&q->campo;
     a=(x<y)?33:44;
 </CODIGO>
]]>
```

En el siguiente ejemplo, la etiqueta más grande es la PELICULA, que contiene al PERSONAL y el ARGUMENTO. A su vez, PERSONAL contiene tanto al DIRECTOR como a los actores (INTERPRETE). Además, hay un comentario con el formato XML.

```
<?xml version="1.0"?>
        <!-- Película mejor valorada en la actualidad -->
        <PELICULA nombre="El Padrino" año=1985>
            <PERSONAL>
                </DIRECTOR nombre=''Georgie Lucar''>
                </INTERPRETE nombre=''Marlon Brando'' interpreta-a=''Don
                    Corleone''>
```

```
                </INTERPRETE nombre=''Al Pacino'' interpreta-a=''Michael
                    Corleon''>
            </PERSONAL>
        </ARGUMENTO descripción=''Película de mafias sicilianas en Estados
            Unidos''>
</PELICULA>
```

3.4.1.3 Definiciones de tipo de documento (DTD's)

XML tiene una ventaja que se puede convertir en un inconveniente: cada persona/autor puede crear sus propias etiquetas.

Crear una definición de tipo de documento (DTD) es como crear tu propio lenguaje de marcado. La DTD define los tipos de elementos, atributos y entidades permitidas, y puede expresar algunas limitaciones para combinarlos. Los documentos que se ajustan a su DTD se denominan "válidos". El concepto de validez no tiene nada que ver con el de estar bien formado. Puede haber documentos XML sin una DTD asociada. En este caso, no son "válidos" ni "inválidos", simplemente "bien formados" o no.

Las declaraciones de tipo de elemento comienzan con "<!ELEMENT" seguidas por el identificador genérico del elemento que se declara. A continuación, se muestra especificación de contenido. Por ejemplo: *<!ELEMENT sistema (varlinguistica, regla)>*

La especificación de contenido puede ser de cuatro tipos:

- EMPTY: Puede no tener contenido. Ejem. <!ELEMENT anglerotation EMPTY>
- ANY: Puede tener cualquier contenido. No es aconsejable usarlo, ya que es conveniente estructurar los documentos XML. Ej. <!ELEMENT anything ANY>
- MIXED: Puede tener caracteres de tipo de datos o una mezcla de caracteres y subelementos. Ej. <!ELEMENT parrafo (#PCDATA | énfasis)*>
- ELEMENT: Sólo puede contener subelementos especificados en la parte de contenido.

Ej. <!ELEMENT esqueleto(cuerpo, extremidades)>

Un modelo de contenido es un patrón que establece los subelementos aceptados y el orden en que se aceptan. Cada partícula puede llevar un indicador de frecuencia, que siguen directamente a un identificador general. Los símbolos aceptados son:

- **coma** (,): Denota secuencia. <!ELEMENT esqueleto(cuerpo, extremidades)> En este caso, el esqueleto debe tener un cuerpo y extremidades.
- **barra vertical** (|): Denota opción.
- **interrogación** (?): Opcional una o ninguna vez.
- **asterisco** (*): Repetible cero o más veces.
- **suma** (+): Necesario y repetible.

Por su parte, las declaraciones de atributos empiezan con "<!ATTLIST". Pueden contener datos de carácter (indicado con "CDATA"). La palabra "#REQUIRED" indica que no tiene valor por defecto, y que es obligatorio especificar el atributo. Se puede omitir el valor de un atributo con "#IMPLIED".

Capítulo 4

Metodología de Trabajo

4.1. Introducción

En este capítulo se detallará el proceso de desarrollo que se ha llevado a cabo para la elaboración del proyecto MOCASYM, haciendo especial hincapié en aquellas fases o módulos más críticos del sistema.

El sistema está formado por varios módulos que se ajustan a las diferentes tareas a realizar desde la entrada de vídeo hasta el resultado final. Cada una de estas tareas se realizará en una fase o etapa específica del sistema, como se puede ver en la Figura A.1

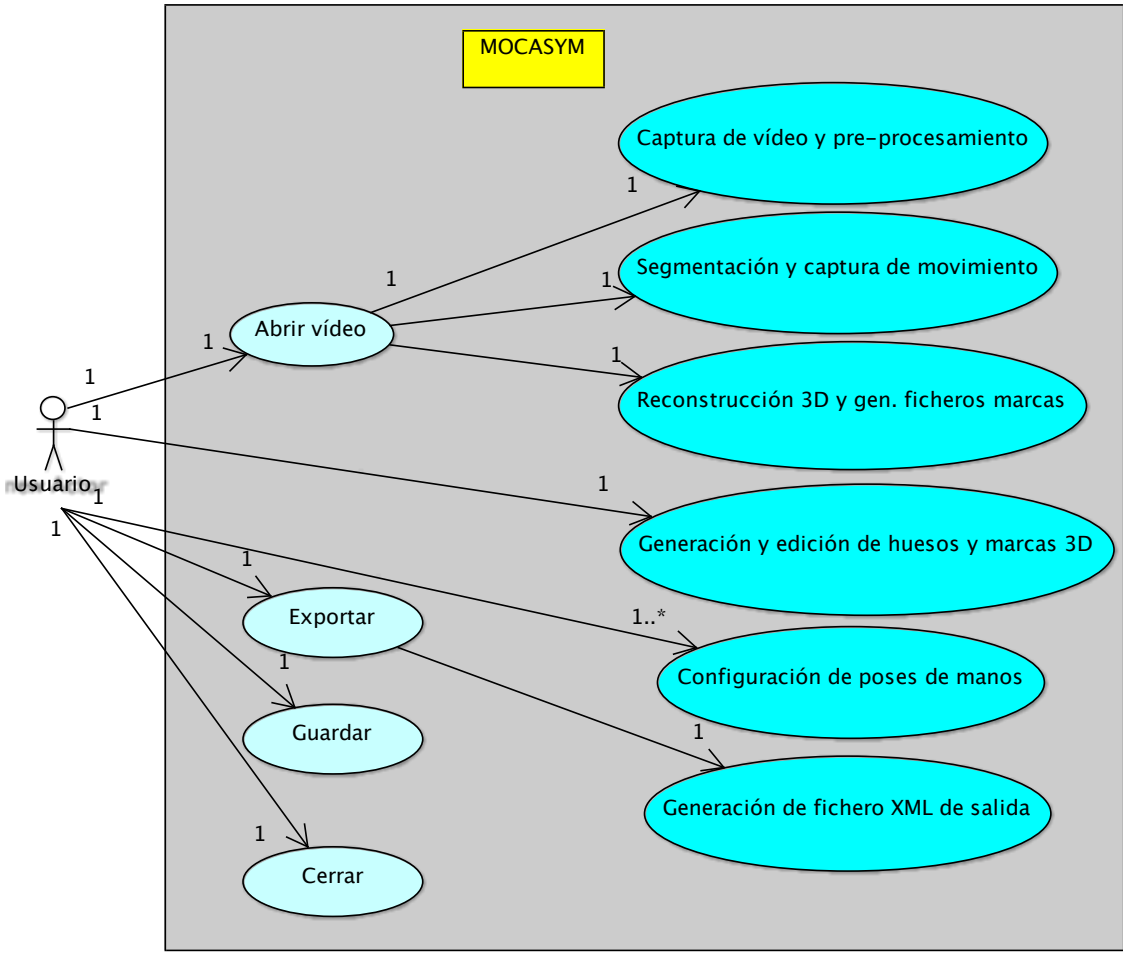


Figura 4.1: *Diagrama de casos de uso general del sistema*

Por lo tanto, las fases en las que se subdivide el sistema son:

- Captura de vídeo y pre-procesamiento de imágenes: en esta fase se elige el vídeo de entrada en el que se signa una palabra determinada, se realiza la captura del mismo y se hace un procesamiento inicial de cada uno de los *frames* capturados.

- Segmentación de imágenes y captura de movimiento 2D: esta fase la compone el módulo fundamental del sistema; se realiza el seguimiento del movimiento del intérprete de

lengua de signos empleando técnicas de segmentación y medición de las imágenes capturadas. Se emplean algunas técnicas de visión por computador como son la umbralización, la detección de bordes y contornos, y algoritmos de *Optical Flow*.

- Reconstrucción del movimiento en 3D y generación de los ficheros de marcas: esta fase es la encargada de transformar las coordenadas capturadas en 2D (se emplea una sola cámara) en coordenadas en 3D y de generar las estructuras de datos necesarios para almacenar los valores calculados. Además, se crearán los ficheros de marcas, con extensiones MF y GFF, para guardar y recuperar la edición de marcas de brazos y cara.

- Generación y edición de huesos y marcas faciales del modelo 3D: en esta fase se genera un espacio tridimensional y se colocan en él los huesos y marcas faciales generadas a partir de las coordenadas calculadas. Se permite modificar esas marcas para perfeccionar y variar los movimientos respecto al vídeo original.

- Configuración de poses de manos: esta fase es la encargada de agregar, a partir de una biblioteca, configuraciones de poses de manos a distintos *frames* del vídeo capturado.

- Generación del fichero XML de salida: por último, en esta etapa se compone el fichero de salida en formato XML con la descripción del movimiento en 3D obtenido como resultado de todas las fases anteriores.

Para el desarrollo de MOCASYM se ha seguido un modelo de desarrollo evolutivo basado en el Modelo de Prototipos. Se han ido construyendo varias versiones del sistema, introduciendo nuevas funcionalidades y modificando otras, hasta llegar a la construcción final que se ajusta a los objetivos generales planteados en un principio.

Se ha seguido esta metodología debido a la naturaleza del proyecto, ya que inicialmente se conocían los objetivos generales del sistema pero no estaban claramente identificados los requisitos de entrada, procesamiento o salida. En ocasiones se han tenido que probar diferentes vías de acción para un mismo problema; el Modelo de Prototipos permite evolucionar paulatinamente la plataforma con las partes más claras y obtener resultados visibles casi inmediatamente para poder probar el resto de módulos en desarrollo. También ofrece un mejor

enfoque cuando el desarrollador no conoce la eficacia de un algoritmo o la forma que debería tomar la interacción de la persona con la máquina.

Los estándares y tecnologías empleadas para el desarrollo del sistema han sido impuestas, en cierto modo, por los objetivos del proyecto. Por este motivo y debido a que las bibliotecas que se consideraban idóneas, tenían su interfaz nativa en C, y puesto que este lenguaje es óptimo, por motivos de eficiencia, para programación gráfica y para funciones relacionadas con el campo de la Visión por Computador, éste es el lenguaje de desarrollo elegido.

Para la interfaz gráfica de usuario se ha empleado GTK (v. 2.12.9), por su buena adaptabilidad con OpenGL[1], y glade (v. 2.6.2), para mantener perfectamente separada la interfaz visual de la lógica y el negocio.

Para tareas de Visión por Computador se han tenido en cuenta varias bibliotecas que cumplen los requisitos impuestos en los objetivos iniciales (ver comparativa en la Tabla 4.1), pero finalmente se han empleado las bibliotecas de OpenCV por ser altamente eficientes y disponer de un marco de desarrollo muy amplio. Además, OpenCV está desarrollado en C/C++, es multiplataforma y cuenta con una licencia BSD (*Berkeley Software Distribution*) compatible con GPL.

Se ha utilizado "libxml2" como biblioteca para el manejo de ficheros XML en su versión 2.6.31.

Se ha empleado un entorno de desarrollo integrado, como es Anjuta en su versión 2.4.1, bajo el sistema operativo GNU/Linux, pero debido al uso de tecnologías multiplataforma es posible la compilación para otros sistemas operativos sin complejidad adicional. Se ha optado por este software porque está perfectamente integrado con el lenguaje de programación C (emplea el compilador gcc y el depurador gdb) y con GTK. Además, Anjuta es software libre y está liberado bajo licencia GPL.

[1] La adaptación se consigue mediante un widget específico: gtkGlExt

	Herramientas de Visión por Computador		
Requisito	**OpenCV**	**Gandalf**	**NeatVision**
Multiplataforma	Sí	Sí	Sí
Procesamiento de imágenes	Sí	Sí	Sí
Análisis de estructuras	Sí	Sí	No
Reconocimiento de objetos	Sí	Sí	Indirecto
Amplitud biblioteca	Muy grande	Normal	Normal
Cantidad de filtros	Muy grande	Grande	Normal
Código abierto	Sí	Sí	Sí
Lenguaje de programación	C/C++	C/C++	Java
Facilidad de uso	Buena	Normal	Muy buena

Tabla 4.1: Comparativa de bibliotecas de Visión por Computador.

4.2. Captura de vídeo y pre-procesamiento de imágenes

Como se ha comentado, esta es la fase inicial en la línea de ejecución del proyecto. Es la encargada de elegir el vídeo de entrada según la petición del usuario (los vídeos de las palabras signadas se encontrarán en el directorio */Videos* de la raíz del proyecto). Las dos tareas principales que se llevan a cabo en esta fase son:

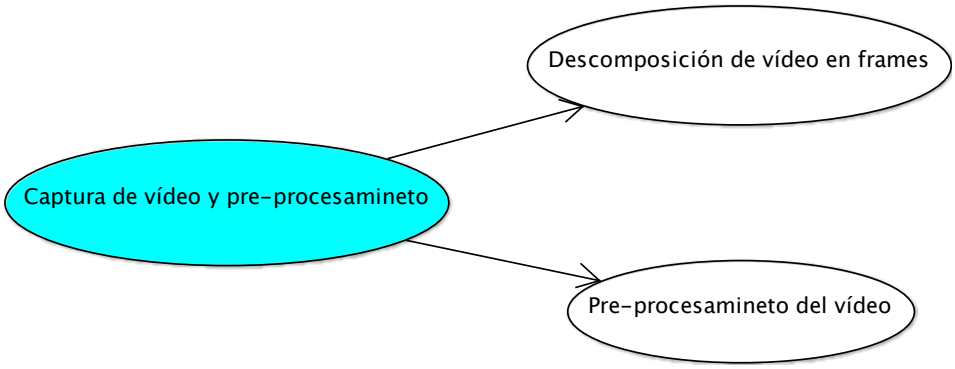


Figura 4.2: *Diagrama de casos de uso del módulo de Captura*

- Descomposición de fichero de vídeo en ficheros de imagen (ver Figura 4.3).

- Pre-procesamiento del vídeo de entrada.

4.2.1. Descomposición de fichero de vídeo en ficheros de imagen

Esta tarea se llevará a cabo de forma automática una vez se haya seleccionado el vídeo de entrada. En la versión del proyecto, para la descomposición del vídeo en *frames* se ha empleado la herramienta de procesamiento de audio y vídeo *FFmpeg* [6], que soporta multitud de formatos de imagen y vídeo.

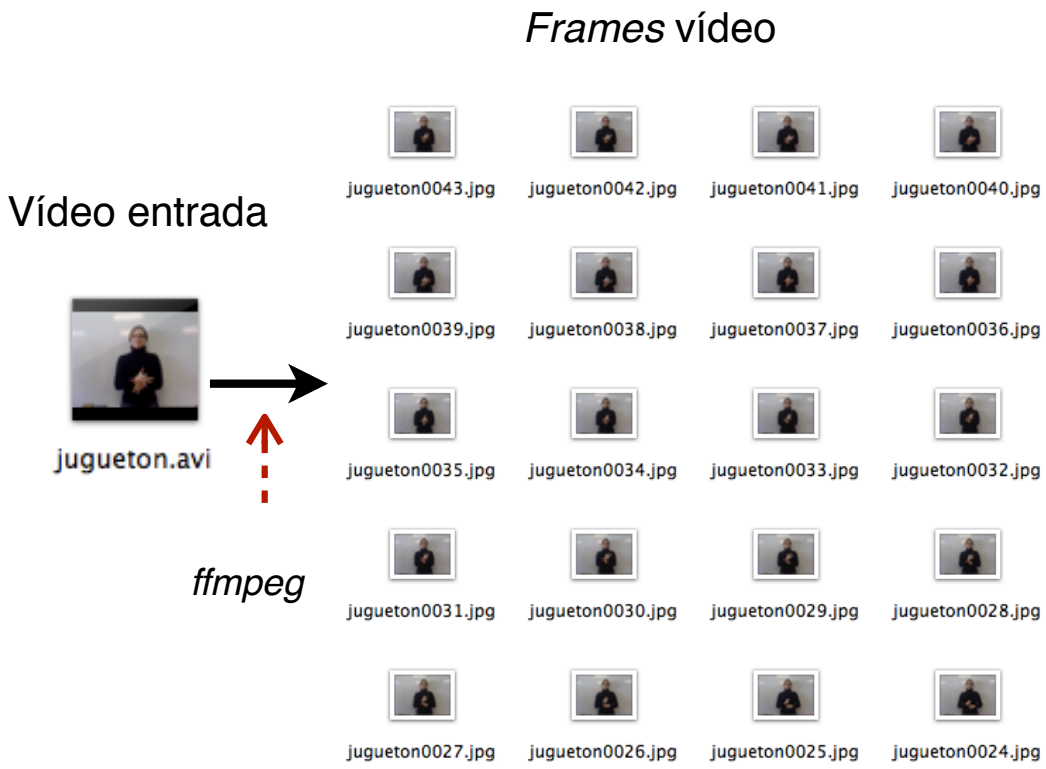


Figura 4.3: *Descomposición del vídeo en frames*

La tasa de *frames* por segundo empleada para la descomposición se obtiene de la información que proporciona el vídeo de entrada. Estas imágenes resultantes se almacenan en un directorio con el nombre del vídeo de entrada dentro del directorio */Images* situado en el directorio raíz del proyecto. El formato de las imágenes generadas es JPG.

4.2.2. Pre-procesamiento del vídeo de entrada

La función principal de esta tarea es la de analizar el vídeo de entrada y crear dos estructuras de datos: una para los movimientos de brazos y otra para los gestos faciales, con información sobre la tasa de *frames* por segundo, número de *frames* totales y número de canales o marcas empleadas para la detección del movimiento. Las estructuras de datos son *TInfoFichero* y *TInfoFicheroGestos*, que están definidas en el fichero *global_var.h* (ver Anexo D).

4.3. Segmentación de imágenes y captura de movimiento 2D

Este módulo es el encargado de la captura de forma automática del movimiento del personaje que signa, a partir de una escena de vídeo, realizando previamente una segmentación y medición de las imágenes capturadas. Para ello, se emplean algunas técnicas de visión por computador como son la umbralización, la detección de bordes, la detección de contornos, la detección de cara, ojos y boca, y la detección del movimiento mediante algoritmos de *Optical Flow*.

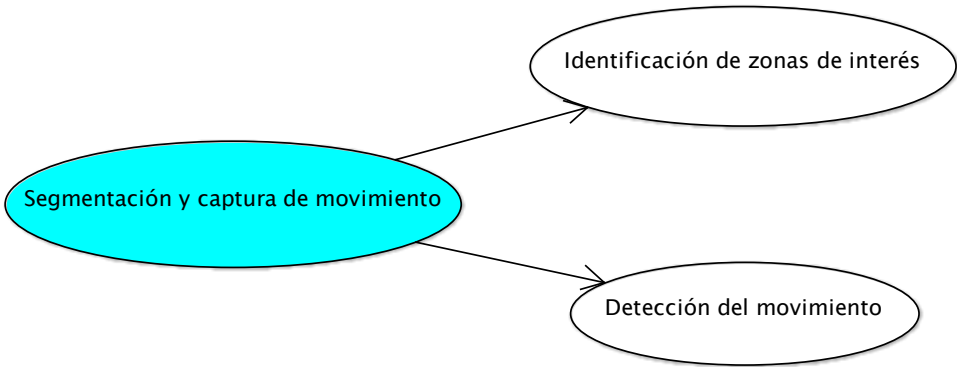


Figura 4.4: *Diagrama de casos de uso del módulo de Segmentación*

Por lo tanto, el módulo podemos subdividirlo en dos tareas (ver Figura 4.4), que son:

- Proceso de identificación de zonas de interés del personaje (ver Figura 4.5).

- Proceso de detección del movimiento con técnicas de *Optical Flow* (ver Figura 4.6).

4.3.1. Proceso de identificación de zonas de interés del personaje

Una vez que el vídeo se ha dividido en *frames* y se ha guardado la información necesaria respecto a ese vídeo, se llama a la función *init_video_process(gchar*)*, encargado de realizar la captura de vídeo; mediante la función *cvCaptureFromAVI* se detecta si hay flujo de vídeo desde archivo, y posteriormente, si el vídeo se ha cargado satisfactoriamente se obtiene y se carga cada *frame* en una estructura de tipo *IplImage** mediante las funciones *cvGrabFrame* y *cvRetrieveFrame* (ver Algoritmo 1).

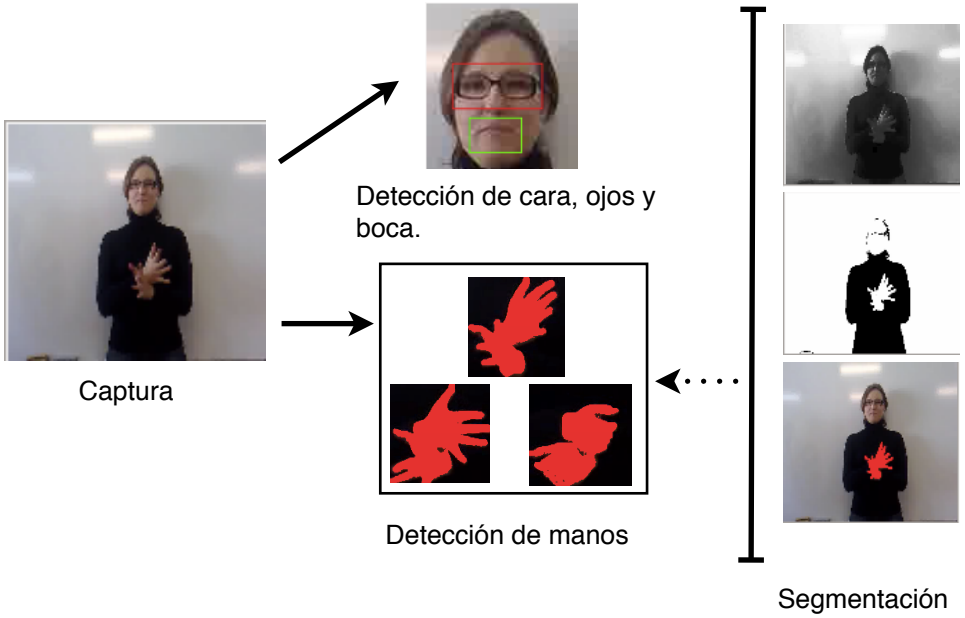


Figura 4.5: *Proceso de pre-procesamiento y segmentación de las imágenes*

Cada uno de los *frames* capturados pasan por un detector de cara, ojos y boca, y se les aplica una detección de bordes y contornos para capturar las manos. Las funciones encargadas de realizar este proceso son:

```
detect_face_eyes_mouth(IplImage*);
find_and_draw_contours(IplImage*);
```

Algoritmo 1 Captura de vídeo

```
1:  CvCapture *capture = NULL;
2:  IplImage *frame, frame_copy = NULL;
3:  storage = cvCreateMemStorage(0);
4:  capture = cvCaptureFromAVI(inputFile);
5:  si (capture) entonces
6:     initializeFaceCapture();
7:     mientras (cvGrabFrame(capture)) hacer
8:        frame = cvRetrieveFrame(capture);
9:        frame_copy = cvCreateImage(cvSize(frame.width, frame.height), IPL_DEPTH_8U,
          frame.channels);
10:       detect_face_eyes_mouth(frame_copy);
11:       find_and_draw_contours(frame_copy);
12:    fin mientras
13:    cvReleaseImage(frame_copy);
14:    cvReleaseCapture(capture);
15: fin si
```

La búsqueda de rostros, ojos y boca se realiza mediante un clasificador *Haar* para cada uno de ellos. Los clasificadores son propios de *OpenCV* y ya han sido previamente entrenados. Tras las búsqueda de cada zona en cada imagen se devuelve un *array* de punteros a estructuras donde se almacena la información de cada zona detectada. Esto permite hacer recortes de imagen mediante selección de Regiones de Interés (ROI), en el caso del rostro, o pintar la zona detectada en cada *frame*, en el caso de ojos y boca.

El algoritmo de detección de rostros (equivalente para el de ojos y boca) emplea una función proporcionada por las bibliotecas de *OpenCv* llamada *cvHaarDetectObjects*, la cual utiliza un clasificador almacenado en un fichero en formato xml para detectar caras en una imagen. En este caso se trata de caras frontales, dado que el clasificador esta entrenado con esa funcionalidad.

Se hace uso de la función *cvCvtColor (const CvArr* src, CvArr* dst, int code)* para transformar la imagen original a escala de grises y la función *cvResize (CvArr* src, CvArr* dst, int interpolation)* para reducirla de tamaño e invocar al clasificador para detectar la cara en este caso, la cual será extraída como región de interés (ROI) para posteriormente detectar la boca y los ojos en una imagen más pequeña y con sólo información del rostro. Para la reducción

de tamaño se utiliza una interpolación bilineal (CV_INTER_LINEAR). Se necesita declarar una estructura estática global *CvMemStorage* * necesaria para almacenar estructuras de datos que crecen dinámicamente como secuencias, contornos, etc... Además, se necesitan declarar las siguientes variables:

```
const char *cascade_name = "haarcascade_frontalface_alt2.xml";
CvHaarClassifierCascade *cascade = 0;
```

donde *cascade_name* es el nombre del fichero xml que contiene el clasificador entrenado para detección de rostros, y *cascade* es un puntero a una estructura de tipo *CvHaarClassifier-Cascade* necesaria para almacenar el clasificador en memoria. A continuación, se carga en memoria el clasificador utilizando la función *cvLoad*, se reserva memoria para la estructura mediante *cvCreateMemStorage* y se llama a la función de detección de rostros que se encarga de convertir la imagen a escala de grises y hacerla más pequeña, para que el clasificador funcione mejor, y llamar a la función *cvHaarDetectObject* para detectar caras (ver Algoritmo 2). Finalmente, se extrae la cara detectada como Región de Interés para la posterior detección de boca y ojos, siguiendo el mismo proceso que para la cara pero tomando como entrada la imagen del rostro y utilizando el clasificador correspondiente en cada caso.

La búsqueda de contornos se realiza para la detección de manos. La función encargada de esta tarea es *find_and_draw_contours(IplImage *frame)* (ver Algoritmo 3)

Primero se transforma la imagen de entrada a escala de grises mediante *cvCvtColor*. Posteriormente se hace una ecualización del histograma para normalizar el brillo y aumentar el contraste de la imagen, seguidamente se somete a la imagen a un proceso de umbralización; esto consiste en seleccionar un nivel *T* que separe dos o más niveles de intensidad de una imagen, así se intenta separar el personaje que signa del fondo de la imagen. La función encargada de realizar esta operación es *cvThreshold*, cuyo valor de umbral seleccionado ha sido *45*, seleccionado tras realizar múltiples ensayos con los vídeos utilizados bajo un entorno controlado definido. A continuación, se procede a la detección de contornos propiamente dicha, con la imagen resultante como entrada de la función *cvStartFindContours*. Mediante la función *cvFindNextContour* se recorren los contornos detectados pero sólo se marcarán como válidos aquellos que superen un área estipulada de 150 píxeles (*HAND_CONTOUR_STIPULATED*),

Algoritmo 2 Detección de rostros

```
1: double scale = FACE_SCALED;
2: IplImage *gray = NULL;
3: IplImage *small_img = NULL;
4: CvSeq *faces = 0;
5: si (!storage) entonces
6:    gray = cvCreateImage (cvSize (img.width,img.height), DEPTH, CHANNELS );
7:    small_img = cvCreateImage (cvSize (cvRound (img.width/scale),cvRound
      (img.height/scale)), DEPTH, CHANNELS);
8:    storage = cvCreateMemStorage (0);
9: si no
10:   cvClearMemStorage (storage);
11: fin si
12: cvCvtColor (img, gray, CV_BGR2GRAY);
13: cvResize (gray, small_img, CV_INTER_LINEAR);
14: faces = cvHaarDetectObjects (small_img, cascade, storage, 1.1, 2,
    CV_HAAR_DO_CANNY_PRUNING, cvSize(HAAR_WIDTH, HAAR_HEIGHT));
15: CvRect* r = (CvRect*) cvGetSeqElem (faces, i);
16: faceCoords[0] = (r.x+r.width) * scale;
17: faceCoords[1] = (r.y+r.height) * scale;
18: faceCoords[2] = r.x * scale + faceCoords[0];
19: faceCoords[3] = r.x * scale + faceCoords[1];
20: cvSetImageROI(img, cvRect(faceCoords[0],faceCoords[1],faceCoords[2],faceCoords[3]));
21: detectEyes(img);
22: detectMouth(img);
```

que es el área algo inferior al área de una mano en la imagen capturada. Aquellas áreas seleccionadas son, en un alto porcentaje, las manos del personaje que signa, por lo que son pintadas en rojo mediante la función *cvDrawContours*. Por último, se liberan recursos mediante la función *cvEndFindContours* para que no haya pérdidas de memoria en este proceso.

4.3.2. Proceso de detección del movimiento con técnicas de *Optical Flow*

En un proceso paralelo al comentado en el apartado anterior se realiza la detección del movimiento de brazos y de gestos faciales (ver Figura 4.6). Para ello se emplea una técnica de Visión por Computador llamada ***Optical Flow***, y en particular, el **algoritmo Piramidal de Lucas y Kanade**. La idea fundamental de esta técnica es que se basa en el movimiento relativo de los píxeles de una imagen cuando la cámara se desplaza con relación a un objeto que se encuentra en la escena enfocada.

Algoritmo 3 Detección de manos

```
1: IplImage *gray = NULL;
2: CvSeq *contours = NULL;
3: int threshold = THRESHOLD_FORTY_FIVE;
4: int size = CERO;
5: si (!imgHF) entonces
6:    imgHF = cvCreateImage (cvSize (img.width,img.height), DEPTH, CHANNELS );
7: fin si
8: si (!storage) entonces
9:    gray = cvCreateImage (cvSize (imgHF.width,imgHF.height), DEPTH, CHANNELS );
10:   storage = cvCreateMemStorage (0);
11: si no
12:   cvClearMemStorage (storage);
13: fin si
14: cvCvtColor (imgHF, gray, CV_BGR2GRAY);
15: cvEqualizeHist (gray, gray);
16: cvThreshold (gray, gray, threshold, 255, CV_THRESH_BINARY);
17: CvContourScanner blobs = cvStartFindContours(gray, storage, sizeof(CvContour),
18: CV_RETR_EXTERNAL, CV_CHAIN_APPROX_SIMPLE, cvPoint(0, 0));
19: contours = cvFindNextContour(blobs);
20: mientras (haya contornos) hacer
21:   size = contours.total;
22:   si (size es mayor que HAND_CONTOUR_STIPULATED) entonces
23:      cvDrawContours(imgHF,contours,CV_RGB(255,0,0),CV_RGB(255,0,0),-1,-
         1,8.cvPoint(0,0));
24:   fin si
25: fin mientras
26: contours = cvEndFindContours(blobs);
```

A partir de esta idea, se realiza la captura del movimiento. Se coloca una serie de puntos o marcas en el ***frame*** **inicial** del vídeo capturado en las zonas donde hay un hombro, un codo y una muñeca de la mano. Como es obvio, ya que es una de las particularidades del sistema para la exención de marcas visuales en la escena, se trabaja en un entorno controlado haciendo corresponder los hombros, codos y muñecas con cada uno de los tercios de la imagen, por lo que la situación de las marcas en esas zonas está controlada a priori. Para la detección de movimientos faciales se espera a que se detecten los ojos y boca, mediante la técnica anteriormente descrita. Una vez detectados, se colocan marcas alrededor de área que delimita la región donde se encuentran en la imagen, que serán los puntos de seguimiento a lo largo de

la captura.

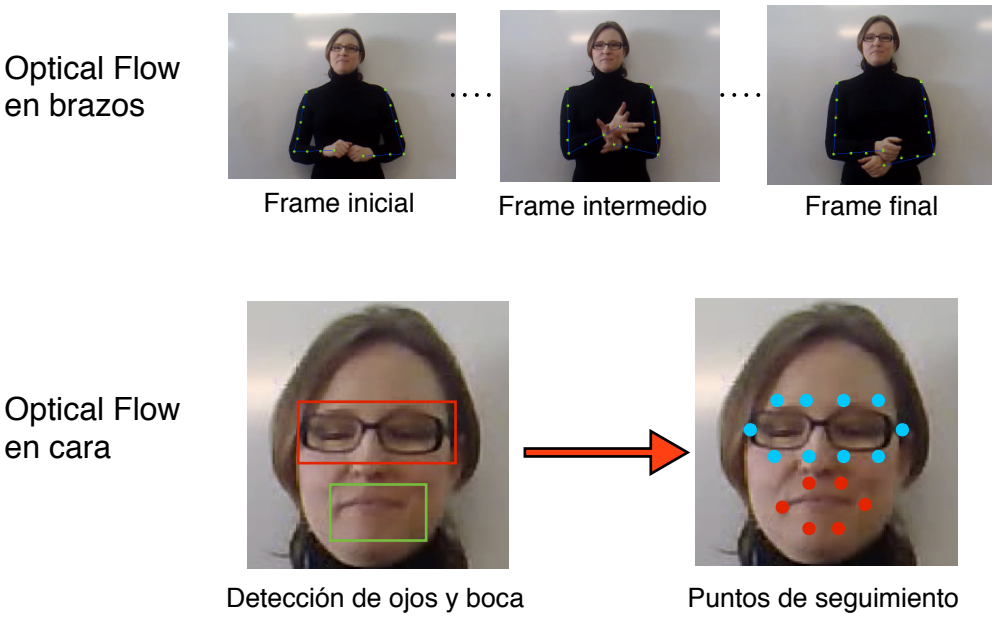


Figura 4.6: *Proceso de medición y captura de movimiento*

Para solucionar el problema que surge con esta técnica cuando un obstáculo se interpone en el camino de uno de los puntos de seguimiento o cuando hay variaciones de brillo por cambios de iluminación, para la detección del movimiento de brazos se sitúan marcas intermedias entre hombros, codos y muñecas (longitudinalmente a lo largo de cada brazo). Así se consigue un mejor control de las marcas desplazadas y/o perdidas.

La función que ofrecen las bibliotecas de *OpenCV* para el seguimiento de puntos es:

```
cvCalcOpticalFlowPyrLK (const CvArr* imgA, const CvArr* imgB,
    CvArr* pyrA, CvArr* pyrB, CvPoint2D32f* featuresA,
    CvPoint2D32f* featuresB, int count, CvSize winSize, int level,
     char* status, float* error, CvTermCriteria criteria , int
    flags ));
```

siendo cada uno de los parámetros:

- **imgA** primer *frame* en escala de grises en un instante *T*.
- **imgB** segundo *frame* en escala de grises en un instante *T* + *dT* (siguiente *frame*).
- **pyrA** *buffer* para el algoritmo piramidal del primer frame.
- **pyrB** igual que pyrA pero para el segundo frame.
- **featuresA** *array* de puntos para ser calculados.
- **featuresB** *array* de puntos 2D calculados para las nuevas posiciones.
- **count** número de puntos calculados
- **winSize** tamaño de la ventana del *frame*.
- **level** Nivel máximo de la pirámide (empleado 3).
- **status** estado para cada punto del *array* de encontrado o no encontrado.
- **error** error permitido en la diferencia del punto original y el movido.
- **criteria** criterio de para cada nivel de la pirámide.
- **flags** flag utilizado para Lucas-Kanade CV_LKFLOW_PYR_A_READY

pero esta función tiene dos principales problemas, que son la pérdida de puntos cada cierto número *frames* capturados y la pérdida o desplazamiento de puntos cuando un objeto se interpone en el camino, como por ejemplo, cuando el movimiento de una mano oculta zonas del brazo que en el *frame* inicial estaban visibles.

Para solucionar el problema se ha implementado una mejora que consiste en generar dos grupos de marcas para cada brazo, un grupo para las marcas entre hombro y codo y otro grupo para las marcas entre codo y muñeca, y controlar en cada *frame* las distancias y ángulos que forman las marcas respecto al resto de marcas de su grupo. Si en un determinado *frame* la distancia de una marca o el ángulo que forma respecto a las otras del mimo grupo es desproporcionado (está fuera de un rango estipulado) significa que esa marca se ha desplazado

o se ha perdido. En caso de ser así, lo que se estima es que la marca desplazada o perdida debe estar a la misma distancia que en el *frame* anterior respecto a las demás marcas de su grupo y en el mismo ángulo en el que están en el *frame* actual (ver Algoritmo 4).

El problema radica en que el posicionamiento de las marcas de la cara no siguen ningún patrón concreto (las marcas no guardan relación de posicionamiento entre sí) y los gestos pueden ser aleatorios.

Algoritmo 4 Control de puntos de seguimiento

```
1:  para (cada frame) hacer
2:     double anguloNew = angle(hombroReferentPoint.x, hombroReferentPoint.y,
       points[1][4].x, points[1][4].y, points[1][i].x,points[1][i].y);
3:     double anguloOk = angle(hombroReferentPoint.x, hombroReferentPoint.y,
       points[1][4].x, points[1][4].y, points[1][2].x,points[1][2].y);
4:     double distanceOld = distance(points[0][4].x, points[0][4].y, points[0][i].x,
       points[0][i].y);
5:     double distanceNew = distance(points[1][4].x, points[1][4].y, points[1][i].x,
       points[1][i].y);
6:     si (distanceNew mayor o igual a (distanceOld + 2) ó distanceNew menor o igual a
       (distanceOld - 2)) entonces
7:        points[1][i].x = points[0][i].x;
8:        points[1][i].y = points[0][i].y;
9:     fin si
10:    si (anguloNew mayor que anguloOk + THRESHOLD_THREE_CENTS ó anguloNew
       menor que anguloOk - THRESHOLD_THREE_CENTS) entonces
11:       calculateCoordinates(points[1][4].x,points[1][4].y,anguloOk,(i-5) * longBrazoIzq-
          do / 4 ,i, points);
12:    fin si
13: fin para
```

4.4. Reconstrucción del movimiento en 3D y generación de los ficheros de marcas

En la fase anterior se ha visto cómo se realiza la captura de movimiento de un intérprete de Lengua de Signos a partir de una escena de vídeo. Los puntos o marcas sobre los las que se realiza el *tracking*, definidas en el primer *frame* de cada vídeo, están definidas en el espacio bidimensional (en coordenadas cartesianas x e y). Sin embargo, para la generación de un es-

pacio tridimensional donde se pueda representar el movimiento capturado, poder modificarlo e interpretar realmente el movimiento, se deben convertir esas coordenadas desde 2D a 3D, definiendo las componentes *x*, *y* y *z*.

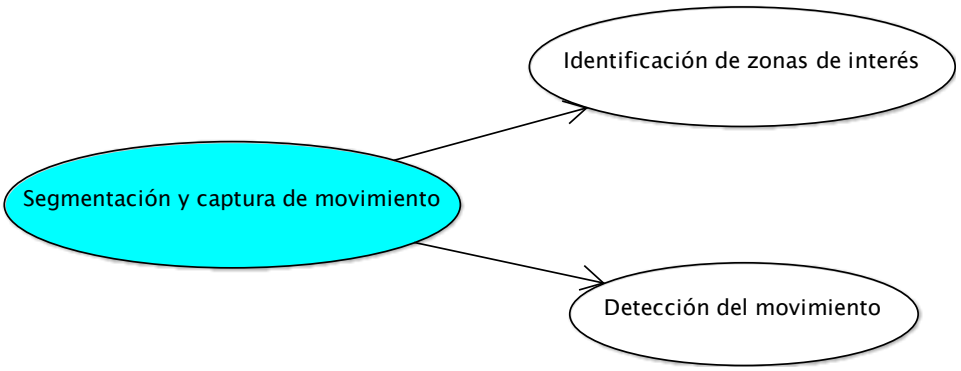


Figura 4.7: *Diagrama de casos de uso del módulo de Reconstrucción 3D*

Esta fase es la encargada de transformar las coordenadas capturadas en 2D en la fase anterior en coordenadas en 3D y de generar dos estructuras de datos para almacenar esos valores calculados, comunes al resto todos los módulos que componen el sistema y que mantienen la consistencia de datos. También se encarga de guardar en un fichero de texto los datos relacionados con la definición de las marcas y los valores de sus componentes *x*, *y* y *z* en cada uno de los *frames* del vídeo de entrada.

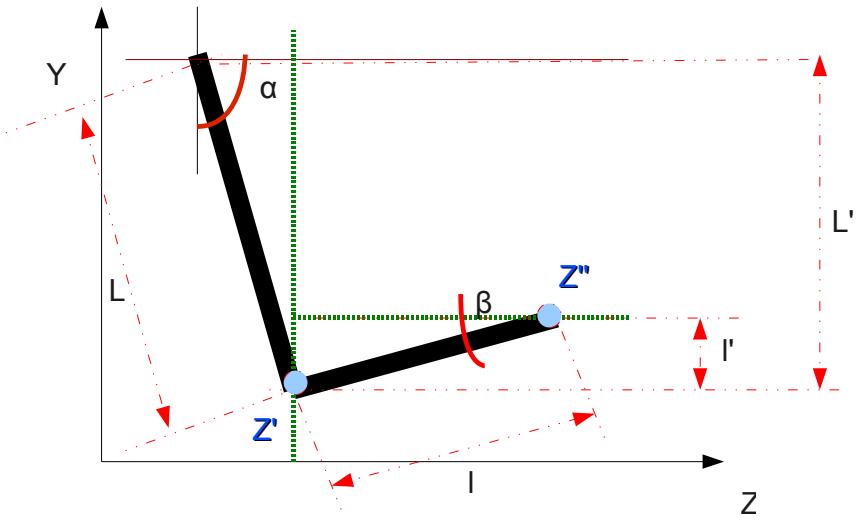
Para comentar el desarrollo y funcionamiento del proceso se subdividirá esta etapa en dos fases:

- Reconstrucción del movimiento capturado en 3D.
- Generación de los ficheros de marcas.

4.4.1. Reconstrucción del movimiento capturado en 3D

El proceso de transformación de las coordenadas 2D a coordenadas en 3D, o lo que es lo mismo, el cálculo de la tercera dimensión (coordenada *z*) se emplean cálculos trigonométricos

a partir del tamaño "real" de brazos (L) y antebrazos (l), que se obtienen en el primer *frame* ya que los brazos parten de una posición situada en un plano paralelo a la cámara, y la longitud "visual" de brazos (L') y antebrazos (l') en cada uno de los *frames* siguientes (ver Figura 4.8).



$$\alpha = \text{acos}(L' / L); \qquad z' = \sin(\alpha) * L$$
$$\beta = \text{asin}(l' / l); \qquad z'' = z' + (l * \cos(\beta))$$

Figura 4.8: *Esquema de cálculo de tercera dimensión*

El Algoritmo 5 resuelve la tercera dimensión empleando restricciones propias del dominio de la aplicación (no se pueden doblar los codos hacia la espalda del intérprete de Lengua de Signos). Una vez calculada, se guardan los datos en un *array* de dos dimensiones a estructuras de datos de tipo *CvPoint3D32f* * que, posteriormente, serán empleadas para generar los ficheros de marcas.

4.4.2. Generación de los ficheros de marcas

Una vez creadas las estructuras de datos con la información de las marcas en coordenadas 3D se crean los ficheros de marcas: uno de marcas de los brazos (fichero MF) y otro de marcas gestuales (fichero GFF). El formato de ambos ficheros es el siguiente:

Algoritmo 5 Cálculo de la tercera dimensión

```
1: mientras (haya marcas de brazos) hacer
2:    double disVista = distance(points[1][i].x, points[1][i].y, points[1][i+2].x,
      points[1][i+2].y);
3:    double longReal = 0;
4:    si (i es la marca muñeca izquierda) entonces
5:       longReal = longAnteBrIzqdo;
6:    si no, si (i es la marca muñeca derecha) entonces
7:       longReal = longAnteBrDrcho;
8:    si no, si (i es la marca codo izquierdo) entonces
9:       longReal = longBrazoIzqdo;
10:   si no
11:      longReal = longBrazoDrcho;
12:   fin si
13:   si (disVista es mayor que longReal) entonces
14:      disVista = longReal;
15:   fin si
16:   double alpha = 0;
17:   double beta = 0;
18:   double zeta = 0;
19:   si ( la marca es un codo o muñeca ) entonces
20:      alpha = acos(disVista / longReal);
21:      zeta = sin(alpha) * longReal;
22:   si no
23:      beta = asin(disVista / longReal);
24:      zeta = points3d[0][i+2].z + (longReal * cos(beta));
25:   fin si
26:   points3d[1][i] = cvPoint3D32f(points[1][i].x, points[1][i].y, zeta);
27: fin mientras
```

```
CHANNELS_SECTION

CHANNEL_TAG     CHANNEL_ID
Nombre_Canal_0_dx   0
Nombre_Canal_0_dy   1
Nombre_Canal_0_dz   2
Nombre_Canal_1_dx   3
Nombre_Canal_1_dy   4
Nombre_Canal_1_dz   5
...                 ...

DATA_SECTION
```

```
CHANNEL_ID      FRAME       VALUE
0               0           valor real
0               0           valor real
0               0           valor real
...             ...         ...
Last_Channel_id 0           valor real
0               1           valor real
0               1           valor real
0               1           valor real
...             ...         ...
Last_Channel_id 1           valor real
0               2           valor real
...
```

Las palabras en mayúscula son identificadores clave de cada parte del fichero de marcas. Se distingue una primera parte de especificación de canales. Cada marca lleva asociada tres canales (uno por coordenada); a cada canal se le asigna un identificador numérico único. Y en la segunda parte del fichero se indica, para cada canal, la posición en el espacio en un determinado *frame*.

El fichero de marcas MF tiene 18 canales correspondientes a las 6 marcas empleadas en la detección del movimiento de brazos, cuya disposición puede verse en la Figura 4.9.

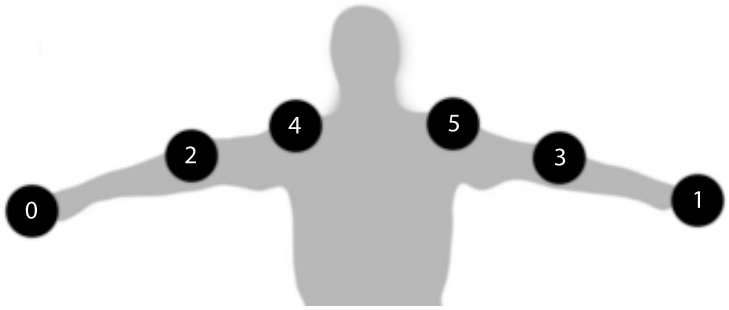


Figura 4.9: *Disposición de las marcas del fichero MF (Marks File) generado.*

El nombre dado a las marcas es libre. Los que se han utilizado en el fichero MF son los de la Tabla 4.2.

El fichero de marcas GFF tiene 48 canales correspondientes a las 16 marcas empleadas en la detección de los gestos faciales y su disposición puede verse en la Figura 4.10.

0 MuncIzda	1 MuncDcha	2 CodoIzdo	3 CodoDcho	4 HombIzdo	5 HombDcho

Tabla 4.2: Nombres de marcas del fichero MF.

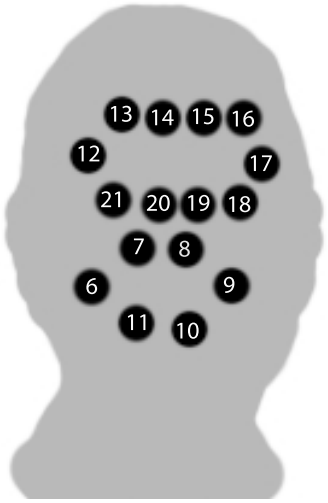


Figura 4.10: *Disposición de las marcas del fichero GFF (Gesture Face File) generado.*

Los nombres que se han utilizado en el fichero GFF son los de la Tabla 4.3. Con estos nombres queda definida la parte CHANNELS_SECTION de los ficheros de marcas.

6 BocaPtIzq	7 BocaSup_1	8 BocaSup_2	9 BocaPtDch
10 BocaInf_1	11 BocaInf_2	12 OjoIzdoCentro	13 OjoIzdoSup_1
14 OjoIzdoSup_2	15 OjoDchoSup_2	16 OjoDchoSup_1	17 OjoDchoCentro
18 OjoIzdoInf_1	19 OjoIzdoInf_2	20 OjoDchoInf_2	21 OjoDchoInf_1

Tabla 4.3: Nombres de marcas del fichero GFF.

4.5. Generación y edición de huesos y marcas faciales del modelo 3D

La etapa que se detallará a continuación sería la primera etapa (no habría que pasar por todas las anteriores al cargar un archivo de vídeo) si el vídeo ya ha sido editado previamente ya que las imágenes de cada *frame* ya estarían creadas y los ficheros de marcas también, y

tendrían los valores correspondientes a los guardados en la última edición del mismo (ver diagrama de secuencia en Anexo A).

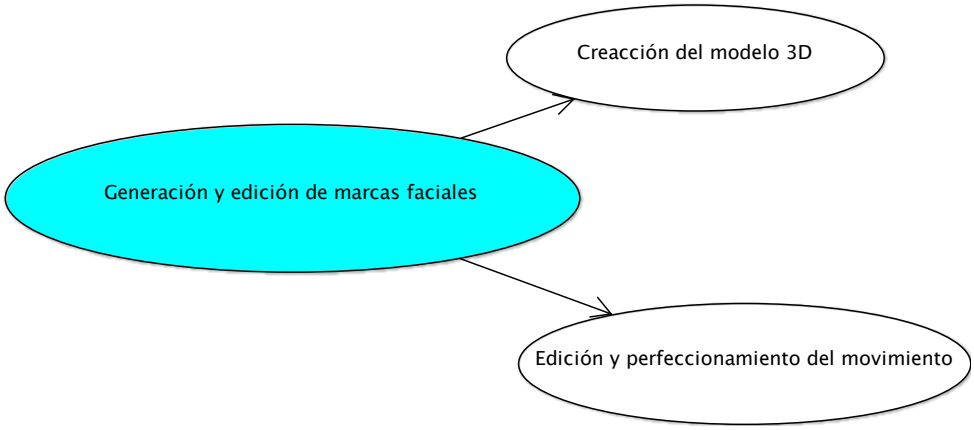


Figura 4.11: *Diagrama de casos de uso del módulo de Generación y Edición 3D*

Una vez alcanzada esta fase, como primera fase o como continuación de la comentada en la Sección 4.4, llega el turno de cargar los datos en una estructura de datos, común a todos los módulos, para continuar con el proceso y generar los huesos y marcas en un entorno en 3D donde simule el movimiento capturado y permita editar y perfeccionar esos movimientos.

Para comentar el desarrollo de esta etapa se subdividirá en dos fases:

- Creación del modelo en 3D.
- Edición y ajuste del movimiento.

4.5.1. Creación del modelo 3D

Para la creación de los huesos y marcas y representarlos en un entorno 3D se necesitan los datos de los ficheros de marcas, que se encuentran cargados en memoria en dos estructuras. Estas estructuras de datos (una para los brazos y otra para la cara) son dos vectores de elementos *TElementoModelo3D* y *TElementoModelo3DGestos* respectivamente. En estos vectores se almacenan toda la información que contienen los fichero MF y GFF en su sección

"DATA_SECTION".

Un modelo3D consistirá en un vector en el que cada elemento es una estructura formada por una cadena de caracteres (el nombre de la marca) y una lista simplemente enlazada. Cada elemento de la lista es una estructura del tipo *TPunto3D*, que contendrá el valor en *x*, *y*, *z* de la marca para cada *frame*. La gestión de la cadena de caracteres y la lista enlazada se realiza mediante la biblioteca *GLib*. Una representación gráfica de esta estructura de datos puede verse en la Figura 4.12

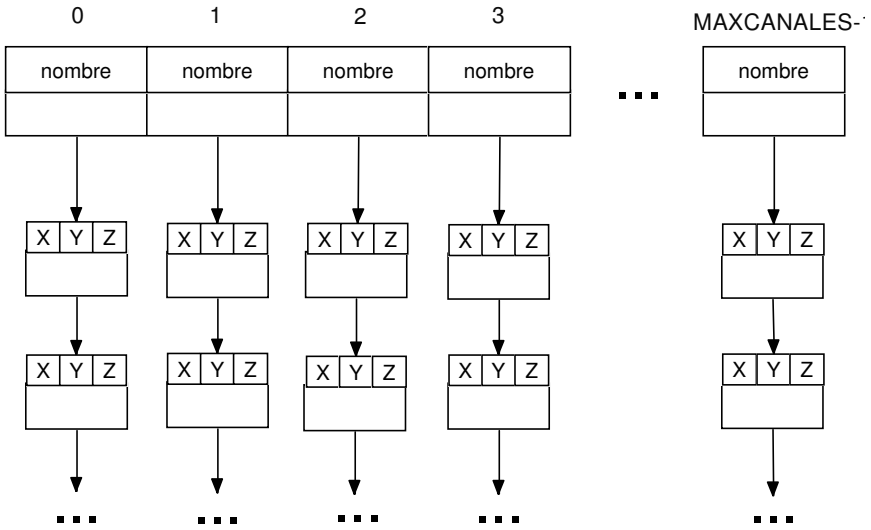


Figura 4.12: *Vector de elementos del modelo 3D de brazos y cara*

La asignación dinámica de la memoria en el almacenamiento del valor de los puntos 3D es un factor implícito si se quiere realizar un sistema genérico. Se deben poder cargar tantos *frames* como memoria disponga el computador. Por tanto, cualquier estimación en un *array* (estático) de valores sería no adecuada.

El resultado de interpretar esos valores, de crear líneas y puntos para la representación de los huesos y marcas, y de crear un entorno en 3D para situarlos, todo ello mediante OpenGL, se puede observar en la Figura 4.13 y la Figura 4.14.

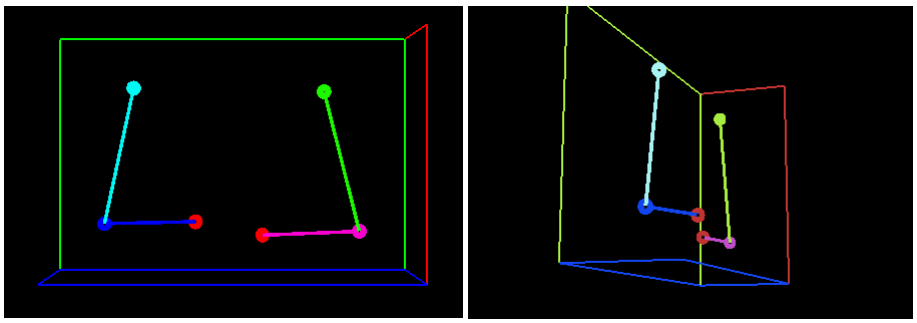


Figura 4.13: *Entorno 3D con el posicionamiento de marcas y huesos de los brazos.*

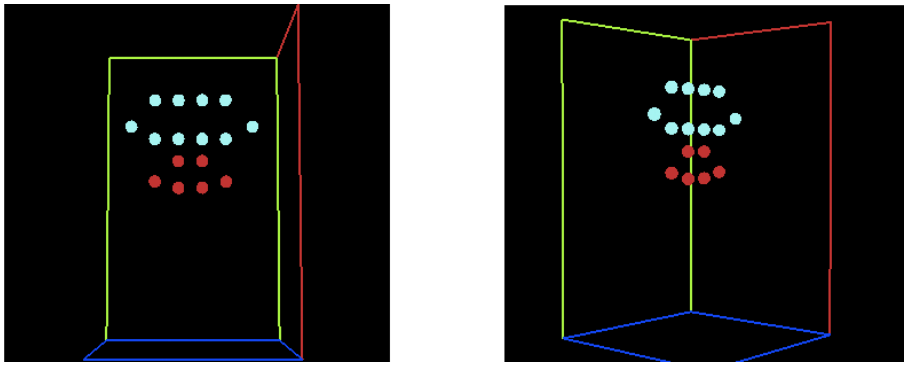


Figura 4.14: *Entorno 3D con el posicionamiento de las marcas de la cara.*

4.5.2. Edición y ajuste del movimiento

Para la edición de marcas se ha empleado un *widget* predefinido de GTK de curvas *spline* basado en las Splines Cúbicas Naturales. Se permite la edición de los puntos de control, que son las marcas definidas anteriormente, sin ningún tipo de limitación. Cuando el usuario quiere confirmar los cambios que ha realizado sobre la curva, se deben extraer los puntos editados y ajustar el valor de los mismos siguiendo la trayectoria de la curva *spline* que se ha definido en el *widget*.

Para utilizar el widget de curvas, los datos tuvieron que escalarse previamente. El *widget* está definido para soportar valores entre 0 y 1. Así, para insertar los valores de un intervalo primero se busca el máximo y mínimo de ese intervalo, y después se aplica la ecuación:

$$valorEscalado = \frac{valorOriginal - minimo}{maximo - minimo}$$

Cuando se obtengan los datos del *widget*, sus valores estarán en el intervalo $[0, 1]$. Para insertar los valores en la estructura de datos, se tienen que "reescalar" a su valor original. Dados como máximo y mínimo los valores calculados antes, sólo hay que despejar *valorOriginal* de la ecuación anterior para obtener:

$$valorOriginal = (valorEscalado \times (maximo - minimo)) + minimo$$

Se buscará, en el vector x de entrada, el valor que se ajuste exactamente al que hay que insertar en la estructura de datos. Como los valores se escalaron (también en el eje x), se empieza en el 0 y se irá sumando $\frac{1}{hasta-desde}$ siendo *hasta* y *desde* los delimitadores del intervalo actual. Así, hasta llegar al 1. Al llamar a la función *inserta_puntos* se asignará uno al último y cero al primer elemento del vector; así se evita que haya puntos de la estructura de datos que no tengan correspondencia en el vector, ni siquiera por interpolación.

Si el valor que se busca en ese momento en el vector x corresponde exactamente con alguno, se le aplica el escalado y se inserta directamente en la estructura de datos. Si no se encuentra el valor exacto (esto sucederá cuando el usuario haya movido el punto), se tiene que interpolar el valor a insertar. Para ello se tomará el intervalo al que pertenece el punto; se calcula su posición dentro del intervalo (valor entre 0 y 1) con la siguiente ecuación:

$$t = \frac{buscado - ValorOrigenIntervalo}{ValorFinIntervalo - ValorOrigenIntervalo}$$

Una vez calculado el valor por interpolación, se inserta y se pasa a buscar el siguiente. Se repetirá el proceso hasta que "buscado" sea 1 (todos los puntos habrán sido introducidos).

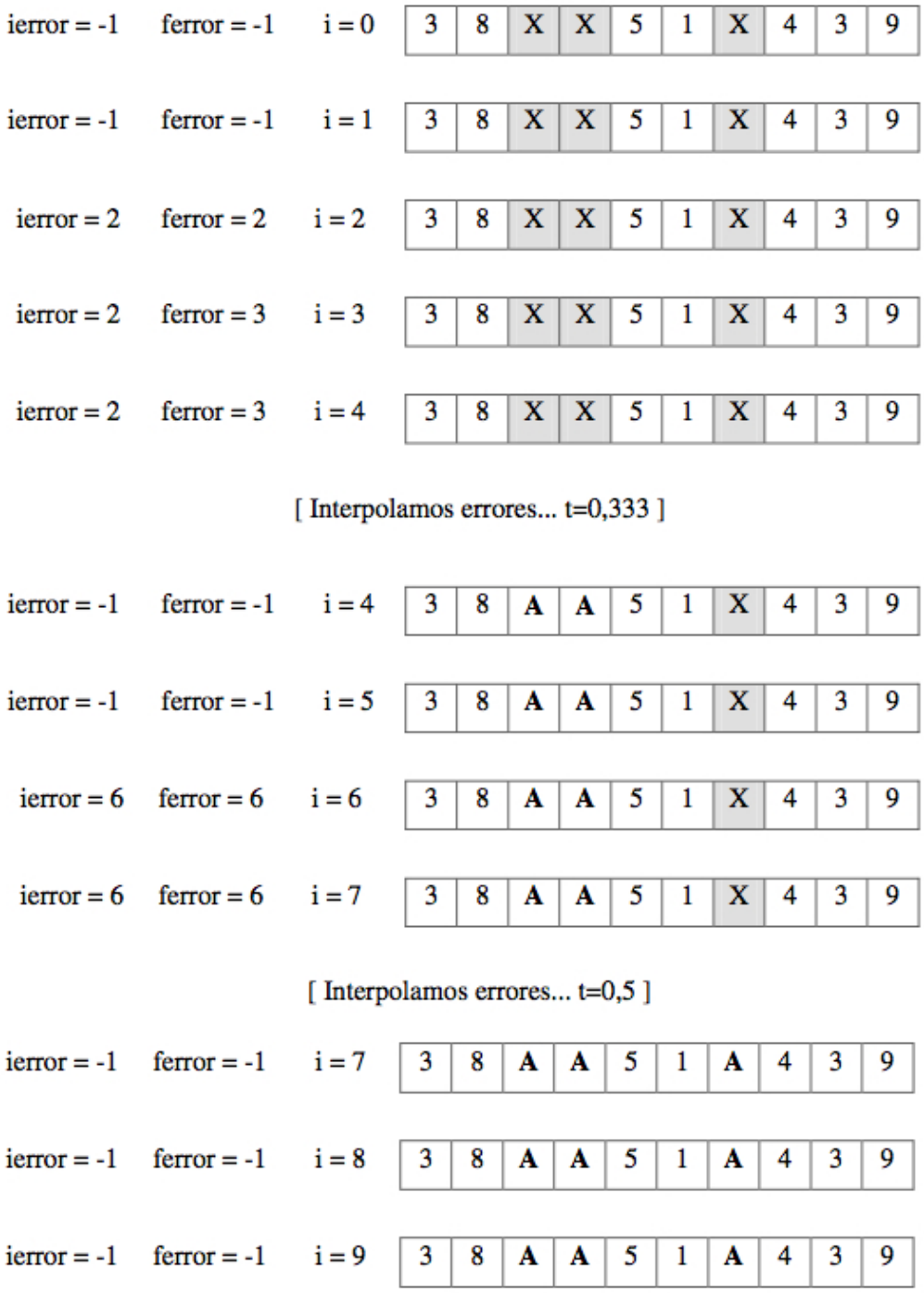


Figura 4.15: *Ejemplo de interpolación de valores erróneos (los valores erróneos se indican con una X y los arreglados con una A).*

La función encargada de la interpolación es la función *interpola*. El prototipo de la fun-

ción recibe dos argumentos:

```
void interpola (int num_puntos, double yo[]);
```

El vector y_0 contiene los valores de los puntos en el eje *y*. El eje *x* no es relevante porque se conocen los valores; todos los valores en *y* estarán separados $1/frames\_seg$ unidades en el eje *x*. El vector *x* será generado en el interior de la función *interpola*.

Se interpolarán de esta forma los valores erróneos del vector. Una traza del algoritmo con $num\_ptos = 10$ podría ser la siguiente (ver Figura 4.15).

Existe otra función de edición que es la función de *suavizado*. La función de suavizado hace uso de la función vista anteriormente. Según el factor de suavizado escogido, se creará un vector lleno de errores, eligiendo un valor correcto cada tantos puntos como indique el factor de suavizado.Por ejemplo, si se tiene el siguiente vector de valores:

35	4	89	43	23	56	42	51	56	90	1	23	53	21	12	80

Si el factor de suavizado escogido es 4, a la función *interpola* se le pasaría el siguiente vector como parámetro:

35	X	X	X	X	56	X	X	X	X	1	X	X	X	X	80

devolviendo esta función el resultado de interpolar los valores erróneos con *splines* cúbicas naturales, que es exactamente lo que se está buscando: un paso suave entre los valores clave del vector:

35	**39**	**43**	**47**	**52**	56	**45**	**34**	**23**	**11**	1	**15**	**32**	**48**	**64**	80

4.6. Configuración de poses de manos

Para la parte de edición de manos se ha tenido en cuenta la escritura alfabética de la Lengua de Signos Española, propuesta en la Universidad de Valencia y aceptada por la CNSE en

su diccionario oficial de LSE.[16].

Aunque al signar los signos se articulan de forma muy rápida, en todos ellos la articulación se realiza llevando a un lugar (del espacio o del cuerpo, y en este caso en contacto o no con esa parte del cuerpo) la mano activa (la derecha en el caso de los diestros) o las dos manos con una configuración determinada, orientándola y, en los signos dinámicos, sometiéndola a un movimiento direccional. Estos elementos no son cualesquiera, sino que en cada lengua de signos hay unos determinados, y diferencian si es monomanual o bimanual, en qué **L**ugar se sitúa, si hay **C**ontacto con el cuerpo, y con qué configuración **Q**, con qué **O**rientación, **D**irección y **F**ormas de movimiento se realiza.

A grandes rasgos, la escritura consiste en recordar estos símbolos y definirlos en orden para cada signo.

Bimanuales		Lugar	Contacto	Configuración	Orientación	Dirección	Forma
S		**L**	(.)	**Q**	**O**	**D**	**F**

Para añadir poses de manos a la captura, la idea radica en centrarse únicamente en el símbolo de la Configuración. Se llama "configuraciones" a las diferentes formas en las que se pueden poner los dedos de la mano. Naturalmente son muchas, pero en la Lengua de Signos no se emplean todas, al igual que en las lenguas habladas no se emplean todos los sonidos articulados posibles.

A partir de esta idea fundamental, y dado que se dispone de un número limitado de configuraciones (31 configuraciones, ver Figura 4.16), aunque podrían sufrir alguna modificación, se ha creado una biblioteca de posiciones de manos con las 31 diferentes configuraciones posibles. De esta manera, el sistema permite añadir poses de manos, diferenciando entre derecha e izquierda, en cada uno de los *frames* donde el usuario vea que la pose ha cambiado.

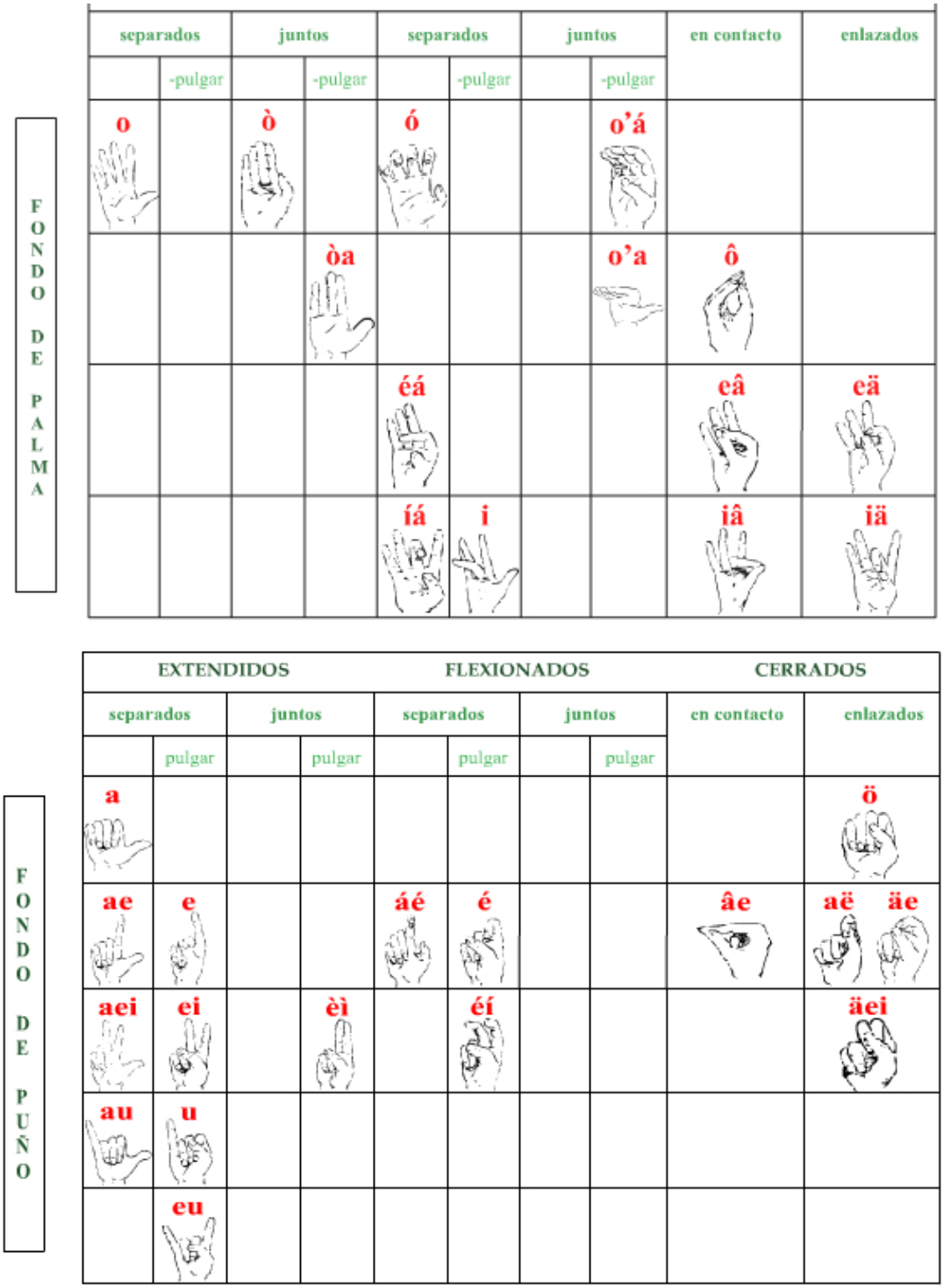


Figura 4.16: *Configuraciones de manos de la Lengua de Signos Española*

4.7. Generación del fichero XML de salida

Por último, este módulo es el encargado de generar el fichero de salida, resultado del proceso automático de captura de movimiento y del proceso manual de edición del movimiento capturado.

El formato del fichero es XML. Contiene la descripción y características del vídeo como son el nombre del vídeo, número de *frames*, tasa de *frames* por segundo, calidad, resolución...

Además de las características comentadas, posee también información de cada una de las marcas de la cara y de los brazos, con los valores de sus coordenadas en 3D en cada *frame*, y con la información de configuración de las poses de manos en aquellos *frames* donde se produce un cambio de la misma (configuración previamente añadida por el usuario en la edición de una captura).

La estructura XML para describir la posición de cada marca en un *frame* es del tipo:

```
<arm_channels>
    <channel channel_name="nombre_de_marca_1">
      <location x="valor_double" y="valor_double" z="valor_double"/>
    </channel>
    <channel channel_name="nombre_de_marca_2">
      <location x="valor_double" y="valor_double" z="valor_double"/>
    </channel>
 .
 .
 .
</arm_channels>
```

La estructura XML para describir la posición de las marcas de la cara en un *frame* es del tipo:

```
<face_channels>
```

```
    <channel channel_name="nombre_de_marca_1">
      <location x="valor_double" y="valor_double" z="valor_double"/>
    </channel>
    <channel channel_name="nombre_de_marca_2">
      <location x="valor_double" y="valor_double" z="valor_double"/>
    </channel>
  .
  .
  .
</face_channels>
```

Y la estructura XML para cada configuración de pose de manos añadida a un *frame* es del tipo:

```
 <hand_channels>
    <channel hand="[derecha|izquierda]">
      <location pose_configuration="escritura_configuración_mano (ej:òa)"/
          >
    </channel>
  </hand_channels>
```

En la Figura 4.17 se observa un ejemplo de archivo XML de salida generado a partir del vídeo de ejemplo “jugueton.avi”.

```
<?xml version="1.0" encoding="UTF-8"?>
<mocasym>
  <word name_word="jugueton"
path="/home/roberto/workspace/MocasymAnjuta/src/Videos/jugueton.avi"/>
  <mvideo>
    <mvideo name="jugueton"/>
    <mvideo framestart="1"/>
    <mvideo frameend="80"/>
    <mvideo fps="25"/>
    <mvideo sizex="640"/>
    <mvideo sizey="480"/>
  </mvideo>
  <action_list>
    <action>
      <action name_action="jugueton"/>
      <action action_start="1"/>
      <action action_end="80"/>
      <pose_list>
        <pose>
          <pose frame_number="1"/>
          <arm_channels>
            <channel channel_name="MuncIzda">
              <location x="-310,142975" y="18,831255" z="-364,285370"/>
            </channel>
            <channel channel_name="MuncDcha">
              <location x="-381,518890" y="0,000000" z="-380,045074"/>
            </channel>
            <channel channel_name="CodoIzdo">
              <location x="-211,274857" y="0,000000" z="-367,808105"/>
            </channel>
            <channel channel_name="CodoDcho">
              <location x="-485,852325" y="0,000000" z="-375,922089"/>
            </channel>
            <channel channel_name="HombIzdo">
              <location x="-242,746048" y="0,000000" z="-221,739548"/>
            </channel>
            <channel channel_name="HombDcho">
              <location x="-447,944794" y="0,000000" z="-225,676620"/>
            </channel>
          </arm_channels>
```

Figura 4.17: *Ejemplo de un archivo XML de salida*

Capítulo 5

Resultados

Este capítulo es una muestra de los resultados obtenidos por MOCASYM en el proceso de captura de movimiento de brazos y movimientos faciales. En las pruebas se han utilizado diferentes vídeos que signan distintas palabras en Lenguaje de Signos Español.

Las secciones en las que se divide este capítulo son:

- **Resultados del proceso de detección de movimientos de brazos:** se muestran los resultados obtenidos en el proceso de captura y detección de movimientos de brazos tras realizar las pruebas y el estudio del dominio de vídeos de ejemplo por parte de un usuario experto.

- **Resultados del proceso de detección de movimientos faciales:** en esta sección se muestran los resultados obtenidos en el proceso de captura y detección de movimientos faciales, detallando en cada caso por qué se producen los errores obtenidos.

- **Tiempos empleados en el proceso de captura y detección de movimientos:** en esta sección se muestran los tiempo para cada una de las etapas del proceso de captura y detección de movimientos del sistema.

Todas las pruebas se han realizado bajo el Sistema Operativo Ubuntu, versión 8.04, en un equipo con un procesador Intel Core Duo a 1,66GHz y 1GB de RAM.

5.1. Resultados del proceso de detección de movimientos de brazos

Para evaluar los resultados obtenidos tras someter al sistema desarrollado a las pruebas con diferentes vídeos de entrada, que representan distintas palabras en lengua de signos, se han tenido en cuenta los *frames* intermedios de cada vídeo, donde el movimiento del actor que signa es más pronunciado.

Para la evaluación se han empleado 5 valoraciones de experto en el dominio de la aplicación. Estas valoraciones son aplicables teniendo en cuenta las trayectorias de, al menos, el 80 % de las marcas, y oscilan entre la valoración *muy mala* de captura y detección del movimiento signado (las trayectorias de, al menos, el 80 % de las marcas difieren mucho de la realidad), a *muy buena*, siendo ésta la valoración máxima, y por tanto, cuando el movimiento detectado se asemeja exactamente al movimiento del actor principal del vídeo (las trayectorias de, al menos, el 80 % de las marcas se asemejan mucho a la realidad).

La Tabla 5.1 y la Tabla 5.2 resumen los resultados obtenidos.

Una vez analizados los resultados obtenidos en el proceso de captura y detección de movimientos de brazos tras realizar las pruebas se han obtenido los siguientes resultados:

- El porcentaje de *frames* donde el 80 % de las marcas de brazos han seguido trayectorias correctas (con valoración *buena* o *muy buena*). Así en los ejemplos estudiados, la captura es válida sin posterior edición manual por parte del usuario en el 73 % de los casos.

- En el 19 % de los *frames* capturados las marcas no se comportan de forma correcta (tienen valoración *regular*, *mala* o *muy mala*).

En la figura 5.1se puede observar la captura de movimiento de brazos de forma gráfica mediante una muestra de dos *frames* al azar para cada vídeo empleado en las pruebas.

	Vídeos de entrada					
Frame	**beber**	**juntos**	**juguetón**	**leche**	**pescado**	**travieso**
25	muy buena	regular	muy buena	buena	buena	muy buena
26	muy buena	regular	muy buena	buena	buena	muy buena
27	buena	regular	muy buena	buena	regular	muy buena
28	mala	regular	muy buena	buena	regular	muy buena
29	mala	regular	muy buena	buena	regular	muy buena
30	mala	regular	muy buena	buena	regular	muy buena
31	mala	regular	muy buena	bueno	regular	muy buena
32	mala	regular	muy buena	bueno	regular	muy buena
33	mala	regular	muy buena	bueno	regular	muy buena
34	mala	regular	muy buena	muy buena	regular	muy buena
35	mala	regular	muy buena	muy buena	regular	muy buena
36	mala	regular	muy buena	muy buena	regular	muy buena
37	mala	regular	muy buena	muy buena	regular	muy buena
38	mala	regular	muy buena	muy buena	regular	muy buena
39	mala	regular	muy buena	muy buena	regular	muy buena
40	mala	regular	muy buena	muy buena	regular	muy buena
41	mala	regular	muy buena	buena	regular	muy buena
42	mala	regular	muy buena	buena	regular	muy buena
43	mala	regular	muy buena	buena	regular	muy buena
44	regular	regular	muy buena	buena	regular	muy buena
45	regular	regular	buena	buena	regular	muy buena
46	regular	regular	buena	buena	regular	muy buena
47	regular	regular	buena	buena	regular	muy buena
48	regular	regular	buena	buena	regular	muy buena
49	regular	regular	buena	buena	regular	muy buena
50	regular	regular	buena	buena	regular	muy buena

Tabla 5.1: Resultados de detección de movimiento de brazos obtenidos en el núcleo de los vídeos (del *frame* 25 al 50).

	Vídeos de entrada					
Frame	**beber**	**juntos**	**juguetón**	**leche**	**pescado**	**travieso**
51	regular	regular	buena	buena	mala	buena
52	regular	regular	buena	buena	mala	buena
53	regular	regular	muy buena	muy buena	mala	buena
54	regular	regular	muy buena	muy buena	mala	buena
55	regular	regular	muy buena	muy buena	mala	buena
56	regular	buena	muy buena	muy buena	mala	buena
57	regular	buena	muy buena	muy buena	mala	buena
58	regular	buena	muy buena	muy buena	mala	buena
59	regular	buena	muy buena	muy buena	mala	buena
60	regular	buena	muy buena	muy buena	mala	buena
61	regular	buena	muy buena	muy buena	mala	buena
62	regular	buena	muy buena	muy buena	mala	buena
63	regular	buena	muy buena	muy buena	mala	buena
64	regular	buena	muy buena	muy buena	mala	buena
65	regular	buena	muy buena	muy buena	mala	buena

Tabla 5.2: Resultados de detección de movimiento de brazos obtenidos en el núcleo de los vídeos (del *frame* 51 al 65).

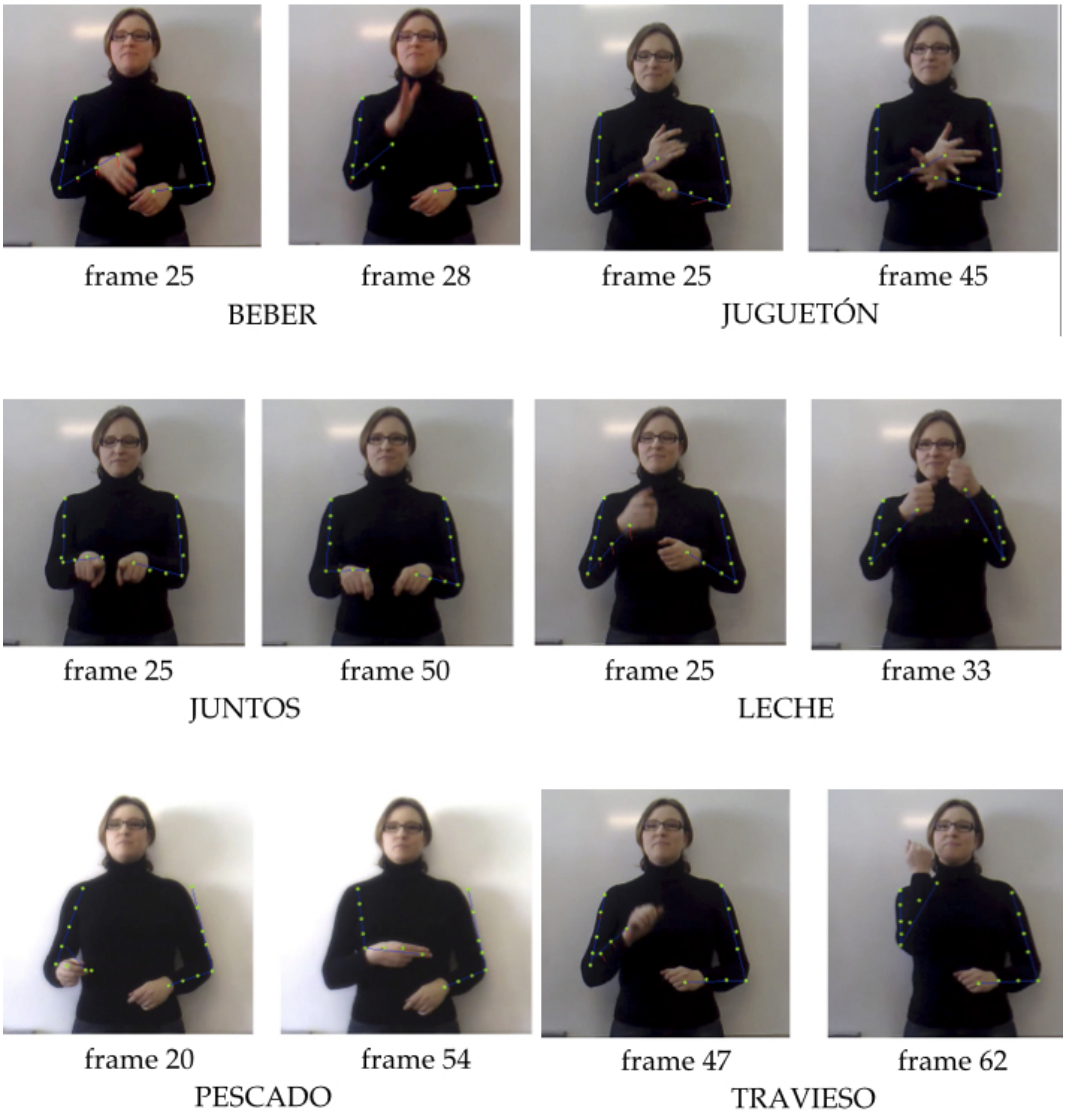


Figura 5.1: Resultados de la captura de movimiento de brazos en una muestra de dos *frames* al azar.

En la mayoría de los casos donde la valoración de la captura es mala o muy mala es debido a la oclusión de puntos de interés por parte del cuerpo del propio intérprete de lengua de signos. El empleo de una segunda cámara desde otro punto de vista podría paliar este tipo de efectos.

5.2. Resultados del proceso de detección de movimientos faciales

En este caso, el intervalo de *frames* para evaluar los resultados obtenidos no se ha podido escoger de forma unánime para todos los vídeos de la muestra ya que la inicialización de las marcas faciales depende del primer *frame* en el que se detectan los ojos y/o la boca. Pero sí se ha evaluado un intervalo de 25 frames, independientes para cada vídeo, a partir del primer *frame* en el que son detectados.

Para obtener los resultados que a continuación se muestran se ha diferenciado entre las marcas encargadas de detectar los movimientos faciales alrededor de los ojos y las dedicadas a capturar movimiento facial alrededor de la boca. De esa forma, se ha calculado una media de las valoraciones obtenidas por un experto en los 25 *frames* estudiados de cada vídeo. Para las valoraciones se emplea el mismo rango que en el caso anterior y se han establecido teniendo en cuenta el mismo criterio: las valoraciones son aplicables teniendo en cuenta que el 80 % de las marcas siguen trayectorias correctas o incorrectas. En la tabla 5.3 se muestran los resultados obtenidos.

	Marcas faciales	
Vídeo	**Ojos**	**Boca**
beber	muy buena	regular
juguetón	muy buena	muy buena
juntos	muy buena	buena
leche	buena	mala
pescado	buena	mala
travieso	buena	muy mala

Tabla 5.3: Media estipulada de los resultados obtenidos en la detección de movimiento facial.

Una vez analizados los resultados obtenidos en el proceso de captura y detección de movimientos faciales, se han estudiado las causas de los errores obtenidos en cada uno de los

vídeos. A continuación, se citan los resultados y se enumeran las causas de los errores obtenidos:

- **beber**
  - Boca: Regular. La causa se debe a la oclusión de zonas de seguimiento por manos y brazos de la persona que signa. Las marcas son desplazadas y no son recuperables.
  - Ojos: Buena.
- **juguetón**
  - Boca: Muy buena.
  - Ojos: Muy buena.
- **juntos**
  - Boca: Buena.
  - Ojos: Muy buena.
- **leche**
  - Boca: Mala. La causa del error se debe a la oclusión de zonas de seguimiento por la mano derecha de la persona que signa. Las marcas son desplazadas y no son recuperables.
  - Ojos: Buena.
- **pescado**
  - Boca: Mala. La causa se debe a una mala detección de la boca con el clasificador empleado, por la insuficiente resolución del vídeo.
  - Ojos: Buena.
- **travieso**

- Boca: Muy mala. La causa se debe a la errónea detección de la boca con el clasificador empleado, por la mala calidad del vídeo.
- Ojos: Buena.

Las conclusiones en vista de los resultados son:

- Los resultados obtenidos en la detección de movimiento en zonas alrededor de los ojos han sido satisfactorios. Las marcas se comportan de forma correcta en un 95 % de los *frames* capturados.
- Los resultados obtenidos en la detección de movimiento en zonas alrededor de la boca han sido algo deficientes. Las marcas se comportan de forma correcta en sólo el 35 % de los *frames* capturados. Las causas se deben, principalmente, a la mala calidad del vídeo o a la pérdida de información de puntos de interés, generalmente, por oclusión debido a la posición de las manos para signar determinadas palabras (ver Figura 5.2).

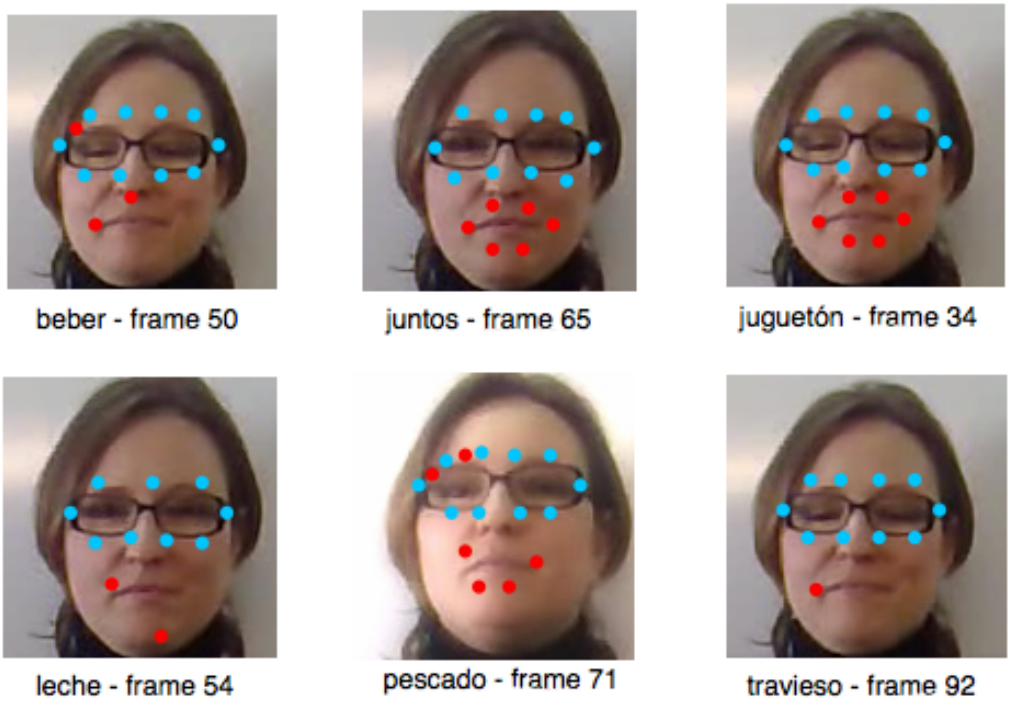


Figura 5.2: Errores obtenidos en la detección de movimiento facial.

Una posible solución al problema sería el empleo de una cámara adicional. Así, con ambas cámaras bien calibradas en distinto ángulos de percepción, esas pérdidas por oclusión de marcas estarían resueltas en la mayoría de los casos, a excepción de que un punto de seguimiento quede oculto en ambas cámaras a la vez en el mismo instante temporal.

También se podría habilitar un control de edición manual para que el usuario especifique la posición de las marcas en aquellos *frames* donde se pierdan las posiciones de las mismas.

5.3. Tiempos empleados en el proceso de captura y detección de movimientos

Los resultados que a continuación se muestran son los tiempos empleados en la edición de los vídeos capturados en el domino de las pruebas por un usuario experto y altamente cualificado en técnicas de animación 3D tradicionales.

Se ha calculado el tiempo empleado en la edición mediante el uso de tres técnicas diferentes (ver Tabla 5.4):

- **Tiempo de generación manual**: Es el tiempo empleado por el usuario experto en animar manualmente el signo correspondiente empleando un esqueleto con soporte de cinemática inversa.
- **Tiempo de captura automática**: Es el tiempo que tarda MOCASYM en capturar y detectar el movimiento de la signante de cada vídeo de entrada, y obtener el primer resultado.
- **Tiempo de edición semi-automática**: Es el tiempo empleado por el usuario de MOCASYM en editar y corregir el resultado inicial, habiendo corregido las trayectorias erróneas e incorporar la información sobre las manos.

En vista a los resultados de la tabla 5.4, si se suma los tiempos empleados en la Captura automática y en la Edición semi-automática, y los comparamos con los tiempos empleados en la Generación manual, obtenemos los resultados de la Tabla 5.5:

	Vídeos de entrada					
Tiempos	*beber*	*juntos*	*juguetón*	*leche*	*pescado*	*travieso*
Generación manual	**82 min.**	**148 min.**	**85 min.**	**82 min.**	**97 min.**	**128 min.**
Captura automática	**2,733 s.**	**3,219 s.**	**2,389 s.**	**3,322 s.**	**4,697 s.**	**3,811 s.**
Edición semi-automática	**4 min. y 12 s.**	**2 min. y 31 s.**	**1 min. y 58 s.**	**4 min. y 39 s.**	**3 min. y 02 s.**	**1 min. y 47 s.**

Tabla 5.4: Comparativa de los tiempos empleados en el proceso de captura y edición de movimiento.

	Vídeos de entrada					
	beber	*juntos*	*juguetón*	*leche*	*pescado*	*travieso*
MEJORA	**94,83 %**	**98,26 %**	**97,65 %**	**94,26 %**	**96,97 %**	**98,56 %**

Tabla 5.5: Resultados de mejoras mediante el empleo de MOCASYM.

El tiempo de mejora que se produce mediante el empleo de MOCASYM para generar la animación (en este caso, descripción del movimiento) de la persona signante respecto al empleo de la técnicas tradicionales basadas en edición de esqueletos para generar animaciones, es del 96,75 %, es decir, con el empleo de MOCASYM se tarda en promedio un 3,25 % del tiempo empleado en la edición mediante técnicas tradicionales.

Capítulo 6

Conclusiones y Propuestas

6.1. Conclusiones

En esta sección se comenta en qué medida se han alcanzado los objetivos propuestos al inicio del desarrollo:

- **Captura automática del movimiento 3D a partir de una única cámara sin necesidad de marcas activas o pasivas. El proyecto, empleando las restricciones del dominio de la aplicación, capturará de forma automática la mayor cantidad de información del movimiento que le sea posible a partir de vídeo real.**

  El sistema desarrollado es capaz de capturar y detectar de forma automática manos, movimientos faciales y movimientos de brazos con un alto nivel de acierto y con ausencia total de marcas en este proceso. En este último caso, se ha resuelto de manera muy eficiente. Esto se ha logrado gracias al empleo *OpenCV* como biblioteca específica que

implementa multitud de algoritmos de Visión por Computador, que ha permitido usar algoritmos potentes de tratamiento de imágenes y de seguimiento de zonas de interés, que unido a las mejoras implementadas hace que el marco de trabajo sea una herramienta potente para la captura de movimientos de brazos y en la detección de rostros, ojos y boca.

La captura de movimiento de manos es el punto más débil debido a la alta complejidad de este requisito, y a la falta de resolución y calidad del vídeo en las zonas de interés para tal efecto. Se ha añadido un interfaz que permite añadir posiciones clave de alto nivel (según especifica el estándar SEA) en fotogramas específicos del vídeo de entrada. Así, se añade un soporte de alto nivel para el movimiento de las manos que podrá ser refinado posteriormente empleando una herramienta de animación 3D.

Teniendo en cuenta los resultados globales, el objetivo se ha resuelto satisfactoriamente.

- **Construcción de una interfaz gráfica de usuario con herramientas de edición avanzada adaptadas al dominio de la aplicación, que permitan editar y reparar movimientos asociados a las capturas anteriores. Esta interfaz de usuario permitirá igualmente añadir información adicional, difícilmente identificable únicamente con un flujo de vídeo, sobre el signo realizado (como el posicionamiento de manos).**

  Este objetivo se ha cumplido plenamente. Se ha desarrollado una interfaz robusta, asegurando que el usuario no pueda realizar acciones no soportadas de forma inadvertida. Para ello, en función del estado en el que se encuentre el sistema, se habilitarán o inhabilitarán en la interfaz las opciones que no sean soportadas en ese momento.

  La interfaz a su vez es flexible y resulta cómoda, ya que permite mostrar u ocultar paneles de herramientas dependiendo de las necesidades de cada usuario.

  En relación a las posibilidades de edición, la herramienta permite la corrección y perfeccionamiento de errores típicos en captura de movimiento. La mayoría de las operaciones a realizar son bastante automáticas, como por ejemplo, interpolar marcas perdidas o suavizar trayectorias. La modificación manual de las trayectorias de las marcas se efectúa mediante el movimiento *drag and drop* del ratón sobre las *Splines* que definen

la trayectoria de cada una de las marcas a los largo de todos los *frames* capturados.

Por otro lado, Se ha desarrollado un espacio tridimensional en el que se representa el modelo 3D, ofreciendo una mejor visualización por parte del usuario para pulir movimientos y trayectorias defectuosas. Es bastante potente; se permite mover y rotar la cámara en cualquier ángulo y sentido, estando o no la animación en ejecución.

Además, se ha desarrollado un panel de edición, siendo una de las funcionalidades la de agregar posiciones predefinidas de manos en el *frame* deseado, a partir de una biblioteca gráfica de poses, de forma fácil e intuitiva.

- **Generación de ficheros de salida independientes del actor virtual, esqueleto o software de aplicación que haga uso de ellos. Este subobjetivo permitirá reutilizar los gestos definidos en MOCASYM en diferentes paquetes software de traducción.**
  Tras el proceso de captura, detección del movimiento, edición de trayectorias y adicción de poses de manos predefinidas, es posible exportar toda la información obtenida a un fichero de salida en formato XML. En él, además de la definición de marcas y poses de manos, se agrega información del vídeo. Así, queda preparado para una posible ampliación de funcionalidad y poder usar un motor de *render* para crear un actor virtual que represente el movimiento definido en el fichero de salida.

- **Construcción del sistema multiplataforma basado en estándares abiertos y tecnologías libres. Para conseguir una amplia difusión en la comunidad de usuarios y facilitar la construcción de signos, el sistema se desarrollará para que pueda ser portado en diferentes arquitecturas empleando licencias libres.**
  Tanto las tecnologías utilizadas (OpenCV. OpenGL, GTK y glade) como los lenguajes de programación utilizados permiten que el sistema sea multiplataforma y no debería haber problema para ser ejecutado en diferentes Sistemas Operativos realizando mínimos cambios.

  Además de esto, toda la tecnología utilizada está publicada bajo alguna licencia de código abierto reconocida por la *Open Source Iniciative* [9] (compatible con GPL), y por este motivo, todo el software desarrollado durante el proyecto se distribuirá bajo licencia *GNU Public License*.

6.2. Líneas de investigación abiertas

Debido a que MOCASYM es la primera aproximación a un sistema de captura de movimiento para la representación de signos de la lengua española, con una sola cámara y con ausencia de marcas, hay muchos aspectos del sistema que podrían ser investigados con mayor profundidad. Aquí se presentan algunas de las ideas que podrían mejorar el sistema y dotarlo de una mayor funcionalidad.

6.2.1. Módulo de detección de poses de manos

Para la implementación de un sistema de captura de movimiento de manos sería interesante emplear un conjunto de sensores flexibles acoplados a cada una de las articulaciones de los dedos de las manos, junto con un microcontrolador y multiplexor analógico (ver Figura 6.1). El principio de funcionamiento consistiría en medir el ángulo de flexión de cada uno de los sensores. El multiplexor analógico permitiría canalizar las señales provenientes de los sensores hacia el puerto de conversión analógico/digital (A/D) del microcontrolador.

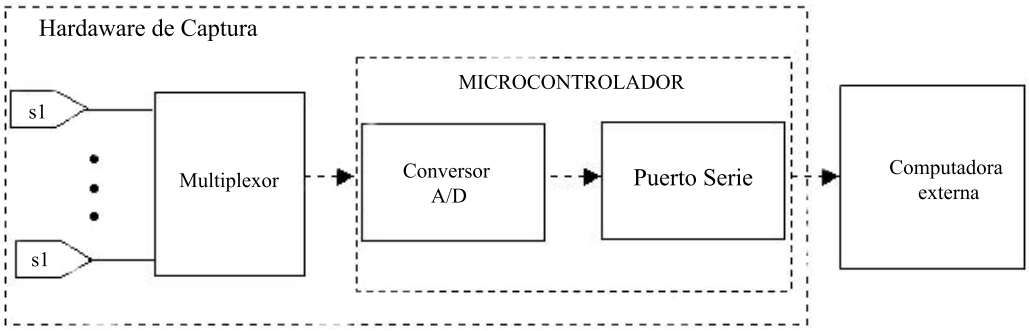


Figura 6.1: *Esquema general del hardware empleado para la captura de movimiento de manos mediante sensores de movimiento.*

Según vayan siendo las señales digitalizadas se iría construyendo un vector donde cada elemento de éste representaría la posición y rotación de cada uno de los dedos de las manos. Pero esta solución presenta un problema en cuanto a costes de implementación, ya que sería necesario calibrar los sensores e implementar un módulo a muy bajo nivel para convertir esas

señales en información de posición y rotación de cada uno de ellos.

Actualmente, hay una alternativa que permitiría disminuir estos costes. Ésta sería el empleo de *Data Gloves* o guantes de datos. El principio fundamental de funcionamiento sería muy parecido al anterior, con la ventaja de que el fabricante provee al usuario el SDK para su manejo y ayuda a la implementación. El proceso de captura de movimiento consistiría en sincronizar la captura de movimiento a través de la cámara con los guantes de transmisión inalámbrica que llevaría puesto en las manos la persona que comunique mediante el lenguaje de signos. De esta manera, el resultado final sería incorporar los movimientos de manos detectados por los *data gloves* a los resultados obtenidos por el sistema MOCASYM. Los costes totales para la implementación de este módulo serían bajos, aunque los costes económicos por hardware aumentarían un porcentaje considerable.

Figura 6.2: *Ejemplo de Data gloves para la captura de movimiento de manos.*

El precio del par de estos dispositivos suele rondar entre 2.000$, con conexión a un computador por puerto serie, hasta los 30.000$, que se conectan mediante red *Wireless* y poseen una alta capacidad sensorial. Un ejemplo podría ser el modelo ´´5DT Data Glove 14 Ultra", del fabricante *Bienetec*. Está basado en 14 sensores flexibles de fibra óptica (dos sensores por dedo más un sensor por nudillo), sensores de 12 bits y conecta a través de puerto USB 1.1. Se vende junto con su SDK, que permite desarrollo bajo Windows y Linux. El precio del conjunto ronda los 3.200$.

6.2.2. Análisis de expresiones faciales

En la Lengua de Signos española, los movimientos faciales cobran especial relevancia ya que acompañan al movimiento de las extremidades superiores a completar de significado un signo. Y en ocasiones, la expresión facial puede cambiar de significado de lo que quiere comunicar la persona signante.

Por este motivo, otra posible mejora del sistema sería la captura de expresiones faciales e integración con los movimientos de las extremidades superiores ya capturados.

Esta meta se conseguiría mediante dos mejoras:

- Aumentando la calidad del vídeo y resolución de aquellas zonas de interés (en este caso, la cara), mejorando así el seguimiento de puntos de interés mediante las técnicas de *optical flow* empleadas en MOCASYM.

- Mediante el empleo de una cámara adicional. Así, con ambas cámaras distantes entre sí, sincronizadas y bien calibradas en distinto ángulos de percepción, esas pérdidas por oclusión de marcas estarían resueltas en la mayoría de los casos, a excepción de que un punto de seguimiento quede oculto a ambas cámaras a la vez en el mismo espacio temporal. Además, este sistema permitiría directamente la captura de movimiento en 3 dimensiones; no sería necesario un proceso de conversión de coordenadas 2D a 3D, como se realiza en MOCASYM.

Esta mejora sería de complicación moderada pero no conllevaría elevados costes temporales, aunque sí tecnológicos.

6.2.3. Mejora del módulo de edición

Uno de los objetivos de MOCASYM era capturar de forma automática el mayor número de movimientos a partir de una escena de video y evitar, en la medida de lo posible, la intervención del usuario en el proceso de edición y reparación de gestos. Este propósito tiene en cuenta que son muchos los usuarios que no saben o no quieren participar en esta tarea, por lo que el proceso de detección del movimiento debería ser mejorado hasta que la parte de

edición fuera necesaria en el menor número de casos posibles.

Así, la mejora del módulo de edición permitirá ahorrar tiempo, incluso en aquellas capturas que requieran poca intervención directa del usuario.

La mejora consistiría en modificar directamente la posición de las marcas en el modelo 3D mediante el uso del ratón o el teclado. Para ello, el primer paso sería seleccionar el objeto en el espacio 3D a mover. *OpenGL* proporciona el modo renderizado GL_SELECTION, aunque es posible utilizar otros métodos. Uno de ellos sería dibujar cada marca con un color diferente, mediante la primitiva *glReadPixels* se podría leer el píxel de la ubicación del puntero del ratón y, examinando el color, se podría determinar qué marca se ha seleccionado. Otra opción sería disparar un rayo desde el puntero del ratón y obtener los objetos en los que interseccciona. Mediante una doble llamada a la función *gluUnProject* desde la posición del ratón, se obtendría el plano cercano (con $winz = 0{,}0$) y el plano lejano (con $winz = 1{,}0$). Con los resultados obtenidos de la plano cercano restados a los del plano lejano se obtendría el vector de dirección XYZ del rayo trazado.

Volviendo a la opción del modo SELECT, para realizar una selección se le debe indicar a *OpenGL* el modo de trabajo mediante:

```
glRenderMode (GL_SELECT);
```

y se debe designar un *array* para ser usado como *buffer* en el que *OpenGL* devuelva la información de selección:

```
GLuint buffer[n];
glSelectBuffer(n, buffer);
```

después se fijaría la transformación de proyección,

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
```

indicando como primera transformación (la última que se realizará), la transformación a la ventana de selección, en torno al cursor. Esto se llevará a cabo mediante la llamada:

```
gluPickMatrix (x, (viewport[3] - y),dx, dy, viewport);
```

siendo (x, y) es el punto en coordenadas de pantalla, obtenidas por el evento del ratón, en torno al cual se quiere seleccionar; (dx, dy) es el tamaño en píxeles de la ventana de selección; y *viewport* es un *array* que contiene el tamaño del *viewport*, y que se puede obtener con la llamada:

```
GLint viewport[4];
glGetIntegerv (GL_VIEWPORT, viewport);
```

Una vez fijada la transformación se vuelve a modo GL_RENDER. Al cambiar el modo se obtendrá como resultado el número de marcas del modelo seleccionadas.

```
hits = glRenderMode (GL_RENDER);
```

Ya solo quedaría (si hits no es cero) analizar la información generada por OpenGL almacenada en el *buffer* y aplicar la translación de las marcas seleccionadas, teniendo como referencia el plano paralelo a la cámara que pasa por cada marca y las coordenadas en el *drag and drop* del puntero del ratón.

Se trata de un proceso que conllevaría unos cálculos de una complejidad media y un coste no muy elevado.

6.2.4. Traductor de Lenguaje de Signos

Otra posible línea de investigación sería la implementación de un traductor de la lengua de signos a lenguaje escrito. La creación de esta herramienta permitiría que una persona que desconoce el lenguaje de signos pueda entenderlo.

Consistiría en traducir el movimiento capturado y definido en el archivo XML de salida del sistema en lenguaje español escrito. Para ello, al igual que GANAS, basado en *Apertium*[1], convierte dinámicamente un texto en lengua de signos, este sistema que se plantea realizaría

[1]Plataforma de código abierto para el desarrollo de sistemas de traducción automática

en proceso contrario: a partir de la lengua de signos se generaría una traducción al lenguaje español escrito.

En este caso, el desarrollo del sistema propuesto conllevaría unos costes muy elevados.

Apéndice A

Diagramas

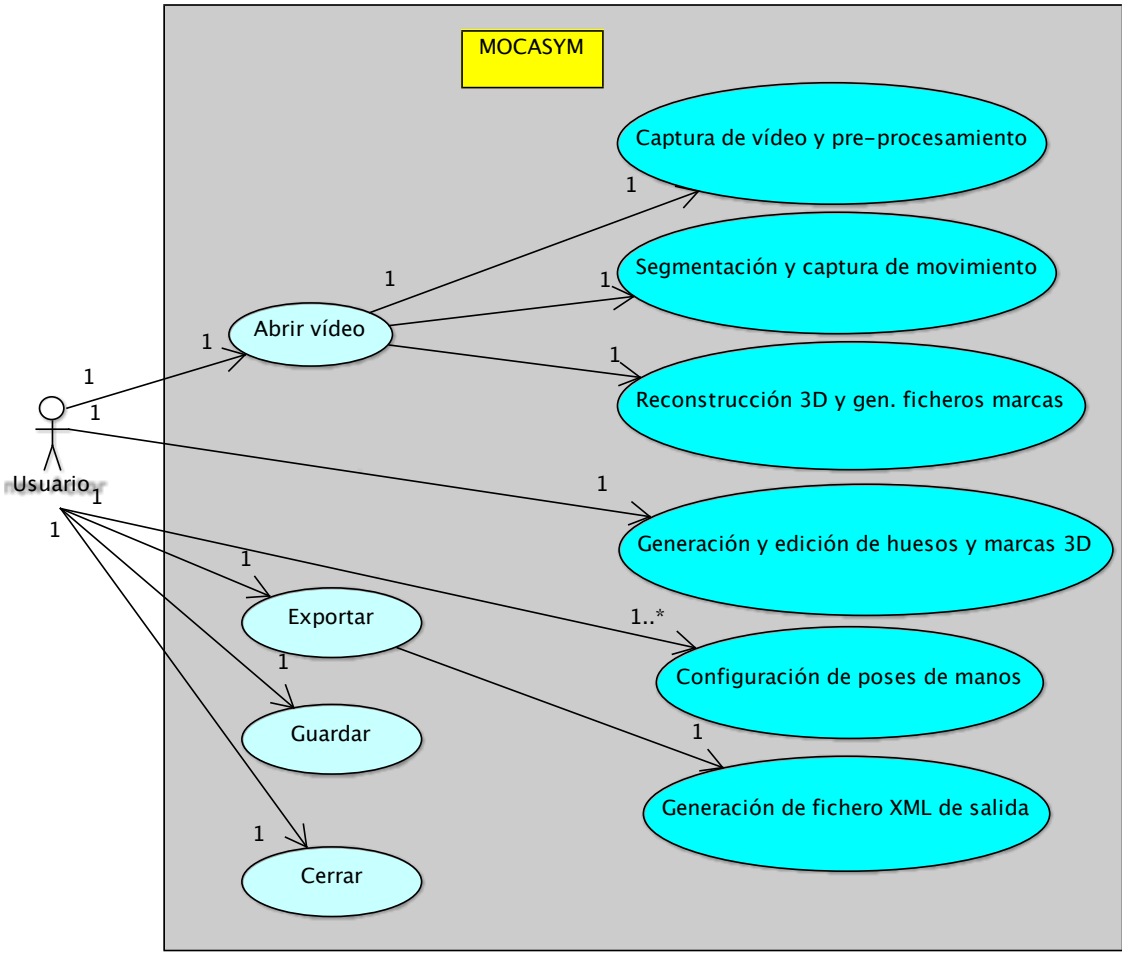


Figura A.1: *Diagrama de casos de uso general del sistema*

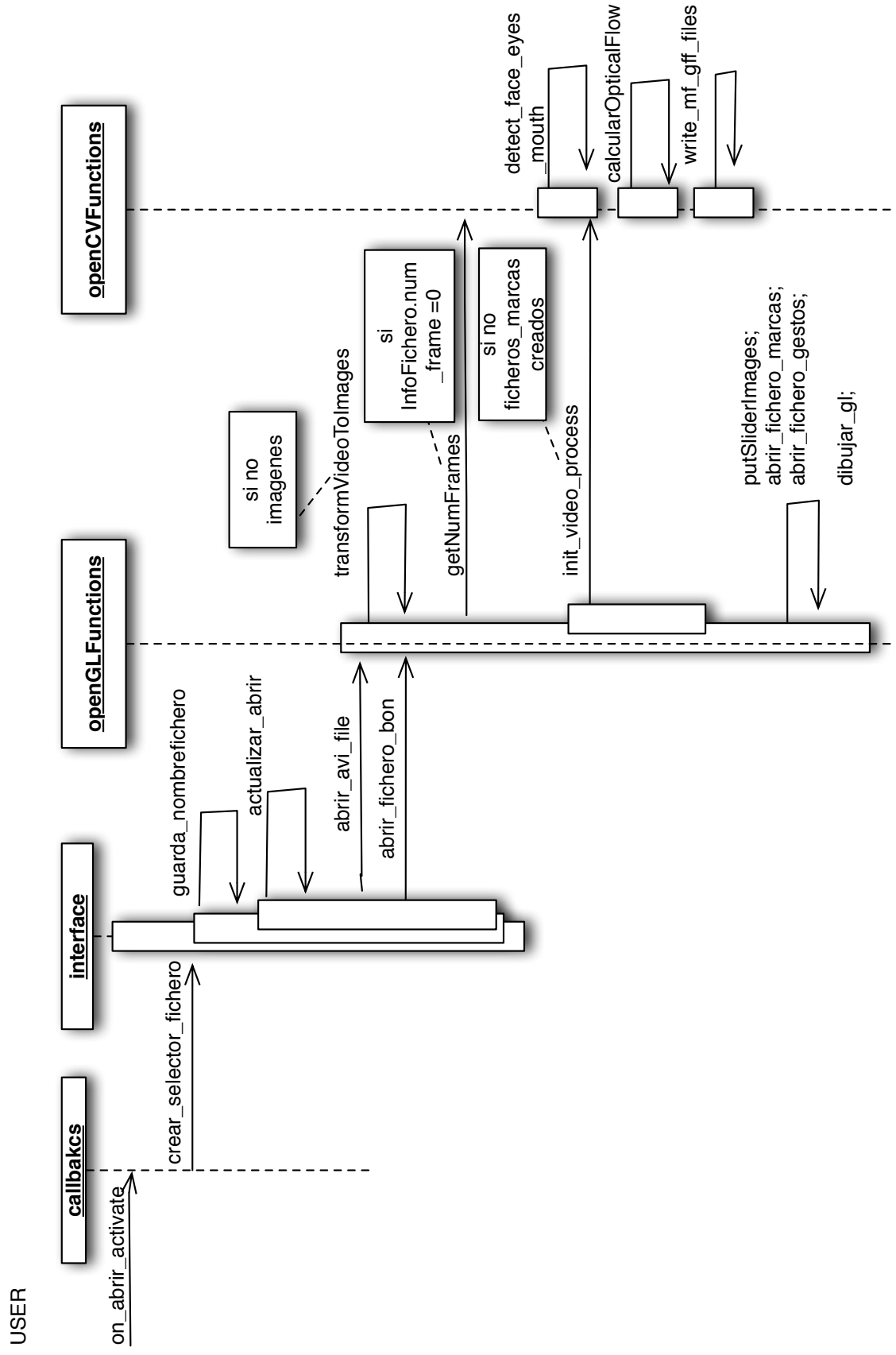


Figura A.2: *Diagrama de secuencia de procesos anterior a la Edición 3D*

Apéndice B

Manual de usuario

En este anexo se presentará la interfaz de usuario de la aplicación y se propondrá un ejemplo, a modo de tutorial, con los pasos a seguir desde que se abre un vídeo para ser procesado, hasta la generación del archivo de salida con los resultados tras la edición de trayectorias erróneas.

B.1. Visión general de la Interfaz de usuario

Una vez ejecutada la aplicación y finalizado todo el proceso de captura y generación de los fichero de marcas MF y GFF, es aspecto de la ventana es la que se muestra en la figura B.1.

En la barra de menús (zona superior de la ventana) se pueden encontrar algunas de las opciones típicas de cualquier aplicación:

- **Archivo:** con las opciones de Abrir un archivo de vídeo, Guardar ficheros de marcas con el nombre del archivo de vídeo procesado, Exportar a XML para generar el archivo de salida, Cerrar trabajo en edición y Salir de la aplicación.
- **Ver:** proporciona las opciones de visualización de los archivos de marcas: o bien para editarlos a mano, o bien para realizar una consulta; Configurar, para que el usuario establezca sus propias preferencias; y las opciones de Mostrar/ocultar el panel de procesamiento de vídeo y el panel de edición.

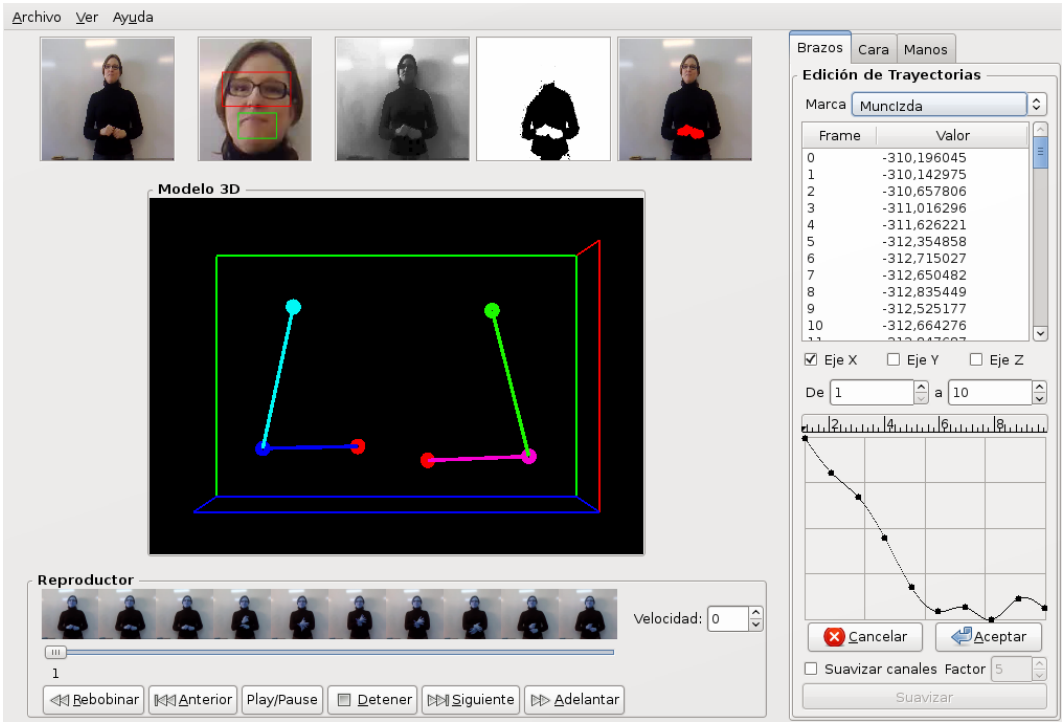


Figura B.1: Aspecto general de la interfaz de usuario cuando un vídeo ya ha sido procesado

- **Ayuda:** con las opciones de Acerca de, con información del proyecto sobre su versión, autores y licencia; y Ayuda, que muestra un manual de usuario similar a éste.

A continuación se muestran cinco imágenes que muestran información de interés a cerca de la captura. De izquierda a derecha, la imagen mostrada es (ver ejemplo en figura B.2):

Figura B.2: Imágenes del proceso de captura en la interfaz de usuario

- **Imagen original:** la imagen mostrada en esta posición corresponde con un determinado

frame de la captura de vídeo (dependerá del estado del reproductor) sin aplicar ningún tipo de modificación en la imagen.

- **Captura de rostro facial:** esta imagen mostrará en todo momento la cara del personaje que signa y, además, en ella se señala la detección de ojos y boca cuando ésta se produce.

Las tres imágenes restantes muestran el proceso seguido para la detección de las manos del personaje que signa:

- **Imagen en escala de grises:** esta imagen es la misma que la original pero con una transformación a escala de grises.
- **Imagen umbralizada:** imagen que representa la anterior imagen en escala de grises aplicándole un determinado umbral para separar tonos de la imagen.
- **Imagen de detección de manos:** esta imagen muestra la imagen original pero con las manos del personaje pintadas en rojo cuando son detectadas en el proceso de detección de contornos aplicado a la imagen umbralizada.

En el marco titulado "Modelo 3D" se mostrará una representación 3D de las marcas del fichero MF o GFF, dependiendo de la pestaña seleccionada en el panel de la derecha (ver ejemplo en figura B.3). La visualización del modelo puede modificarse de los siguientes modos:

- **Rotación:** Pinchando con el botón izquierdo del ratón y arrastrando se rotará la cámara alrededor de una esfera que rodea la escena.
- **Traslación:** Pinchando con el botón derecho del ratón y arrastrando, se trasladará la cámara a otro punto de origen.
- **Zoom:** Para hacer Zoom en el modelo hay que situar el puntero del ratón sobre el mismo y desplazar la rueda del ratón, haciendo *zoom in* o *zoom out*.

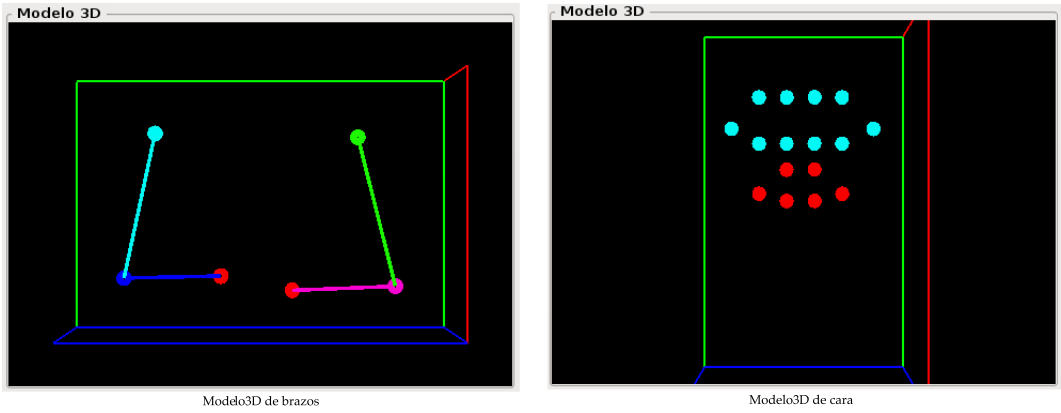


Figura B.3: Aspecto del modelo 3D en la interfaz de usuario

Seguidamente, en la zona inferior de la ventana se muestra el panel de reproducción del modelo 3D, que permite navegar a lo largo del vídeo de forma sencilla e intuitiva (ver figura B.4). Los botones que lo componen son:

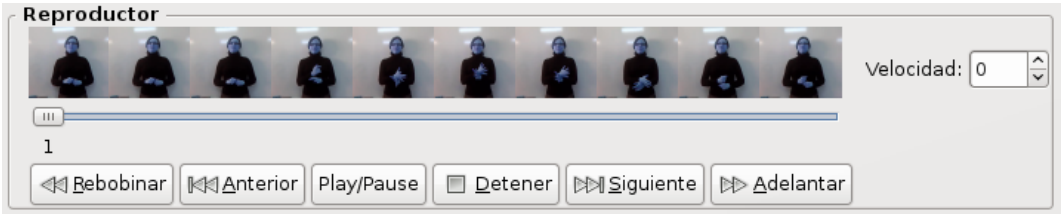


Figura B.4: Reproductor del modelo 3D

- **Rebobinar:** retrocede al inicio de la secuencia de *frames*.
- **Anterior:** retroce un *frame* en la secuencia.
- **Play/pause:** inicia la reproducción si está parada, o la pausa si está en reproducción.
- **Detener:** detiene la reproducción.
- **Siguiente:** avanza un *frame* en la secuencia.

- **Adelantar:** avanza al final de la secuencia de *frames*.

La barra de desplazamiento permite avanzar o retroceder a un *frame* determinado si se pulsa con el ratón y se mueve hacia la izquierda o hacia la derecha de la misma.
Las imágenes en pequeño muestran el *frame* del vídeo en el instante de reproducción que indica la barra de desplazamiento.
El *widget* de **Velocidad** indica la velocidad a la que se reproducirá la animación. Un valor 0 hará que MOCASYM muestre todos los *frames*; un valor positivo indicará el número de *frames* a saltar (el movimiento será más rápido). Un valor negativo indicará el retardo que se aplicará al mostrar cada *frame* (útil en ordenadores rápidos o para analizar más detenidamente el movimiento).

La edición de los datos se realiza desde el conjunto de paneles de la zona derecha (ver figura B.5).

- El **Panel de Brazos** permite modificar manualmente la trayectoria de cada canal correspondiente a una marca de brazos. Para ello, se selecciona una marca del menú desplegable y un intervalo de *frames* (De - A). La lista inferior muestra el valor que toma el canal para cada *frame*. Debajo se encuentra la zona de edición de trayectorias. Cada punto es la posición del canal seleccionado en un *frame*. Se puede calcular rápidamente la posición de cada *frame* por la regla situada en la parte superior. Se pueden eliminar puntos de la curva simplemente pinchando y arrastrando hasta una posición que esté ocupada por otro punto, o bien modificar su situación. También se pueden añadir puntos a la curva pinchando en alguna parte de la trayectoria que esté libre. Por último, está la opción de Suavizar, produciendo un suavizado de trayectorias de todas marcas con el factor de suavizado que se indique en el *widget* Factor.

- El **Panel de Cara** permite modificar manualmente la trayectoria de cada canal, esta vez correspondiente a una marca facial. El funcionamiento es similar al panel de Brazos exceptuando que para las marcas de la cara no se permite suavizar trayectorias.

- El **Panel de Manos** permite añadir configuraciones de poses de manos a un *frame* determinado. Para ello, se selecciona Mano derecha o Mano izquierda, una pose de la

Librería de Poses de Manos, y se pulsa sobre el botón Añadir. Esa pose seleccionada, se añadirá automáticamente a la Lista de Poses de Manos. El botón Sincronizar actualiza el *frame* seleccionado dependiendo del *frame* señalado en el reproductor en ese momento.

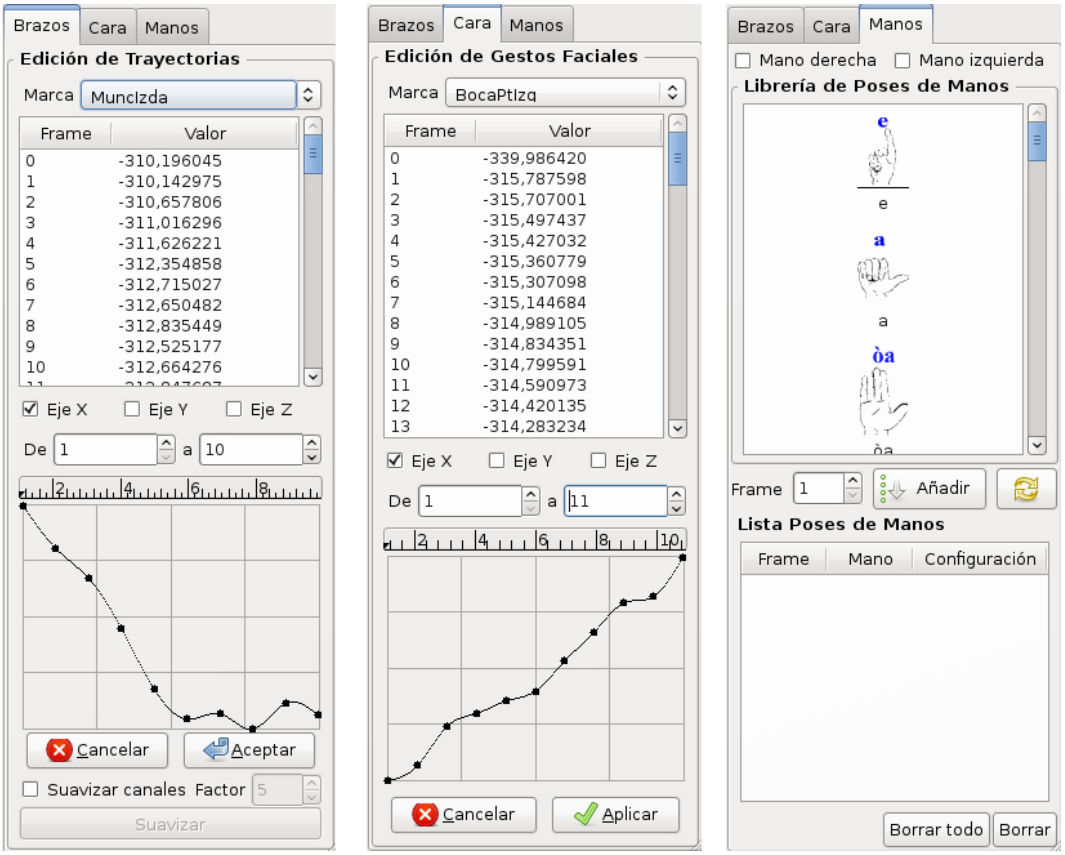


Figura B.5: Los tres paneles de edición de la captura

B.2. Ejemplo de uso de la aplicación

El siguiente tutorial hace uso de las principales opciones de MOCASYM. Para no hacer excesivamente largo el ejemplo, sólo se editará la trayectoria de una marca de brazos, otra de la cara y se añadirán 4 poses de manos en los primeros 4 *frames*. Para finalizar se generará el archivo XML de salida y se mostrarán los resultados.

Paso 1. Abrir archivo de vídeo. Para abrir un vídeo e iniciar automáticamente el proceso de detección de movimiento seleccionar Archivo - Abrir, seleccionar un vídeo contenido dentro del directorio Vídeos y pulsar Aceptar. En ese momento se iniciará automáticamente el proceso de captura y detección del movimiento del vídeo seleccionado.

Paso 2. Familiarizarse con el reproductor y el modelo 3D. Cuando todos los *widgets* que aparecían inhabilitados en la pantalla inicial pasan a estar habilitados, componentes inicializados, imágenes cargadas y el modelo 3D creado, el proceso de detección habrá terminado. En este momento se puede mover el modelo 3D (como se ha comentado en el apartado anterior), reproducir la captura o utilizar los paneles de edición para modificar trayectorias erróneas de las marcas del modelo 3D.

Paso 3. Editar una marca de un brazo. Para editar una trayectoria errónea de una marca de un brazo se procederá al uso del panel de edición "Brazos". Si la trayectoria errónea se produce en un rango determinado de *frames*, establecer el valor de *frame* inicial del intervalo en el *spinbutton* "De" y el final en el *spinbutton* "a". El siguiente paso sería seleccionar la marca a editar en el combo "Marca". Para eliminar puntos y obtener una curva limpia, arrastrar los puntos a eliminar junto a otro que no se desee eliminar o fuera de la curva de interpolación. Una vez eliminados los puntos no deseados, pulsar el botón "Aceptar" y el sistema ajustará el valor de las marcas para situarlos en los *frames* correctos. Se repetirá el proceso para el Eje Y y Eje Z. Si sólo se desea modificar a mano una trayectoria, puede realizarse sin perder los puntos modificados; moviéndolos en sentido vertical y pulsando "Aceptar".

Para suavizar automáticamente la trayectoria, seleccionar "Suavizar canales", elegir un factor de suavizado, y pulsar sobre el botón "Suavizar". Instantáneamente, la curva quedará suavizada.

Paso 4. Editar una marca facial. El procedimiento para editar una trayectoria errónea de una marca facial es similar a la de un brazo, pero desde el panel de edición "Cara" (ver Paso 3). En este caso no se permite suavizar trayectorias.

Paso 5. Agregar poses de manos. Para agregar poses de manos en un determinado *frame*, ir al panel de edición “Manos”. Es hecho imprescindible marcar una sola mano y una configuración de poses de manos en la “Librería de Poses de Manos”. una vez hecho esto, seleccionar el *frame* donde se quiera añadir la pose y pulsar sobre el botón “Añadir”. Para elegir el *frame* actual del reproductor, pulsar sobre el botón “sincronizar”.

Paso 6. Generar fichero de salida. Esta paso sería el final si se desea obtener un resultado. Para ello, ir al menú “Archivo” y seleccionar “Exportar a XML”. El resultado se podrá encontrar en el directorio “Salida” del proyecto, con el nombre del vídeo de entrada seleccionado.

Apéndice C

Manual de Instalación

En este anexo se explicarán los pasos necesarios para ejecutar la aplicación en los Sistemas Operativos más comunes para los usuarios. Para esto se necesitan los siguientes paquetes:

- **GTK+:** Las bibliotecas de GTK, para la interfaz de usuario, pueden descargarse de su sitio oficial:

  `http://www.gtk.org/download.html`

  En la página se encontrarán los pasos necesarios para instalar la biblioteca y cada una de las dependencias de *GTK+*, como son: *GLib*, *cairo*, *Pango*, *ATK*, *fontconfig*, *freetype*, *expat*, *libpng* y la biblioteca *zlib*. Dependiendo del sistema operativo, algunas de ellas se instalarán automáticamente con la biblioteca *GTK+*. Además, es necesario instalar la extensión de *GTK*, como es *GtkGLExt*. Esta biblioteca puede encontrarse en:

  `http://projects.gnome.org/gtkglext/download.html`

- **Glade:** Al igual que *GTK*, para la interfaz de usuario es necesaria la biblioteca *Glade*. Ésta puede descargarse desde su sitio oficial:

  `http://glade.gnome.org/`

- **OpenGL:** El soporte *OpenGL*, por lo general, salvo sistemas muy anticuados, está pensado para trabajar bajo el respaldo de un hardware capaz de realizar las operaciones necesarias para el renderizado, pero si no se dispone de ese hardware, estas operaciones se calcularan por medio de un software contra la CPU del sistema. Así que los

requerimientos hardware son escasos, aunque cuanto mayor sea las capacidades de la maquina, mayor será el rendimiento de las aplicaciones *OpenGL*.

En principio, cualquier versión de windows viene con las bibliotecas necesarias para ejecutar cualquier aplicación que utilice *OpenGL*. Para el desarrollo de las mismas, el Microsoft Visual Studio, y en particular Visual C++ trae también todo lo necesario. En caso contrario, la biblioteca *GLUT* sería necesaria, y puede descargarse desde la página de *Nate Robins*:

`http://www.xmission.com/ nate/glut.html`

En el caso de Linux, para visualizar aplicaciones *OpenGL* es necesario instalar un paquete para el soporte de las bibliotecas de *Mesa* y de las utilidades *GLU* y *GLUT*. Para realizar desarrollo se necesitarán los paquetes equivalentes en modo *dev*.

- **OpenCV:** La biblioteca de visión por computador puede descargarse desde la página el proyecto:

  `http://sourceforge.net/projects/opencvlibrary/`

  No obstante, para las últimas versiones de la biblioteca para sistemas operativos basados en *debian*, pueden surgir complicaciones. Si es así, se recomienda seguir en detalle lo que se comenta en la página:

  `http://www.samontab.com/web/2010/04`
  `/installing-opencv-2-1-in-ubuntu/`

Los archivos fuente ya se encuentran compilados, por lo que la aplicación puede lanzarse:

- Desde la consola de comandos. Dentro del directorio *src* del proyecto se debe escribir:
  - En Windows: *mocasym.exe*
  - En Linux: *./mocasym*
  - En Mac OS X: *./mocasym* ó *open mocasym*
- o bien, haciendo doble clic con el ratón sobre el archivo ejecutable.

Apéndice D

Código fuente

A continuación se detalla la funcionalidad de los módulos que componen el sistema, los cuales pueden encontrarse en el CD-Rom que acompaña a este documento. Debido a la longitud del código fuente (más de 7000 líneas), en este anexo sólo se ha incluido el código de los módulos más relevantes.

D.1. Descripción de los módulos que componen el sistema.

- **/src/global_var.h**: este módulo contiene la definición de tipos de datos y constantes empleadas en la mayoría de los módulos.

- **/src/openGLFunctions.c**: este módulo es el encargado de inicializar los componentes GTK+ de la interfaz gráfica de usuario y de realizar todas las tareas relacionadas con el "modelo3D" y que hacen uso de la biblioteca gráfica OpenGL.

- **/src/openCVFunctions.c**: módulo que contiene aquellas funciones encargadas de realizar tareas relacionadas con la Visión por Computador y hacen uso de las bibliotecas de *OpenCV*. Las funciones más relevantes contenidas en él son las encargadas de realizar el seguimiento de puntos de interés mediante el algoritmo de flujo óptico, conversión de coordenadas 2D a 3D, detección de cara, ojos y boca, y la función para la detección de manos.

- **/src/splines.c**: en este módulo se encuentran implementados los algoritmos de interpolación y suavizado de marcas, basados en interpolación mediante *splines* cubicas naturales.

- **/src/exportFunctions.c**: contiene todas aquellas funciones que trabajan con la biblioteca *libxml2*. Se encarga de generar el fichero XML de salida.

- **/src/interfaz.c**: contiene funciones que realizan tareas relacionadas con la interfaz gráfica. Inicializa y actualiza *widgets* mediante el empleo de bibliotecas de *GTK+* y *glade*.

- **/src/callbacks.c**: módulo que contiene todas aquellas funciones relacionadas con el manejo de eventos de la interfaz gráfica.

- **/src/xmlParseConstants.h**: contiene las constates empleadas en el módulo de generación del XML de salida: *exportFunctions.c*

- **/src/utils.c**: módulo destinado a funciones que realizan operaciones de cálculo matemático necesarias en otros módulos, como operaciones aritméticas y trigonométricas.

- **/src/callbacks.h, /src/interfaz.h, /src/utils.h, /src/splines.h, /src/headers.h** y **/src/openGLFunctions.h**: son ficheros que contienen las cabeceras de los módulos que llevan su mismo nombre.

D.2. Código fuente de los módulos más relevantes.

global_var.h

```
/*********************************************************************
 *   UNIVERSIDAD DE CASTILLA-LA MANCHA     ESCUELA SUPERIOR DE INFORMATICA
 *
 *   Proyecto Fin de Carrera: MOCASYM: Sistema de captura semi-automática
 *   del movimiento para la representación de la Lengua de Signos Española
 *
 *   Autor: Roberto Mancebo Campos  Fecha: 17 de Septiembre de 2010
 * -------------------------------------------------------------------
 *   Fichero: global_var.h
 *   Contenido: Definicion de tipos de datos y constantes utilizados por
 *    la mayoria de los modulos.
 *********************************************************************/

#define MAXCANALES 350 /* Numero maximo de canales que podemos tener */
#define MAX_PTOS 4000  /* Numero maximo de puntos de control */
#define MAX_HUESOS 50  /* Numero maximo de huesos para el esqueleto */
```

```
17 #define NOMBRE_TEMPORAL "#mocasym_temp#.###"
18 #define NOMBRE_TEMPORAL_GESTOS "#mocasym_temp_gestos#.###"
19
20 #define VISTA_BRAZOS  0
21 #define VISTA_CARA   1
22 #define VISTA_MANOS  2
23
24 typedef struct {  /* La informacion que interesa de un fichero huesos */
25   int frames_seg;
26   int num_canales;
27   int num_frames;
28 } TInfoFichero;
29
30 typedef struct {  /* Informacion que interesa de un fichero de gestos */
31   int gestos_frames_seg;
32   int gestos_num_canales;
33   int gestos_num_frames;
34 } TInfoFicheroGestos;
35
36 typedef struct {   /* Un punto en 3D se define con 3 coordenadas */
37   double x;
38   double y;
39   double z;
40 } TPunto3D;
41
42 typedef struct {   /* Elemento del vector de canales de huesos */
43   GString *nombre;
44   GSList *lista;
45 } TElementoModelo3D;
46
47 typedef struct {        /* Elemento del vector de canales de gestos */
48   GString *gestos_nombre;
49   GSList *gestos_lista;
50 } TElementoModelo3DGestos;
51
52 typedef struct {        /* Elemento del vector de poses de manos */
53   GString *frame_clave;
54   GString *mano;
55 } TPoseMano;
56
57 typedef struct {        /* Elemento del vector de canales de manos */
58   GString *mano_pose;
59   GSList *mano_lista;
60 } TElementoModeloPoseManos;
61
62
63 typedef struct {    /* Informacion para cada marca del esqueleto */
64   int color;         /* Numero de color */
65   gboolean visible; /* Es una marca visible */
66 } TMarcaEsqueleto;
67
68 typedef struct {    /* Informacion para cada hueso del esqueleto */
69   int color;         /* Numero de color */
70   int origen;        /* Marca origen del hueso */
71   int destino;       /* Marca destino del hueso */
72 } THuesoEsqueleto;
73
74 typedef struct {    /* Informacion de cada color del esqueleto */
75   int numero;
76   char nombre[30];
```

```
77    int r;            /* Componente roja del color */
78    int g;            /* Componente verde del color */
79    int b;            /* Componente azul del color */
80  } TColorEsqueleto;
```

Funciones más relevantes de *openGLFunctions.c*

```
1   /**********************************************************************
2    *     Funcion: calculaProyeccion
3    * Descripcion: Calcula en la matriz de proyeccion; situando el
4    *            origen en el punto donde debemos dibujar.
5    *    Entradas: Ninguna.
6    *     Salidas: Ninguna.
7    **********************************************************************/
8   void calculaProyeccion(void)
9   {
10   float dx, dy, dz;   /* Dimensiones de la caja que vamos a usar*/
11   float diagonal;     /* Diagonal del suelo de la caja */
12   float distCamMundo; /* Distancia de la camara al origen */
13   float zCerca, zLejos; /* Determinan los planos de recorte en Z */
14   float relacAspecto;  /* Relacion entre el ancho y alto del Widget */
15
16   glViewport(0,0,ventanaAncho,ventanaAlto);
17   glMatrixMode( GL_PROJECTION );
18
19   glLoadIdentity();
20
21   /* Dimensiones de la caja que vamos a usar */
22   dx = maxx - minx;
23   dy = maxy - miny;
24   dz = maxz - minz;
25
26   diagonal = sqrt(dx*dx + dy*dy + dz*dz);
27
28   /* Usamos el doble de la diagonal de la caja, para
29    que la camara nunca entre en el mundo durante la rotacion. */
30   distCamMundo = 2*diagonal;
31
32   /* distCamMundo es ahora la distancia desde la camara al
33   origen del mundo. Estas distancias son escogidas para que
34    el mundo este situado ENTRE los planos de recorte en Z. */
35   zCerca = distCamMundo - 1.00001*diagonal/2;
36   zLejos = distCamMundo + 1.00001*diagonal/2;
37
38   /* Trasladamos el plano de imagen. */
39   glTranslatef(-2*trans[0]/ventanaAncho,-2*trans[1]/ventanaAlto,0);
40
41   /* Escalamos las coordenadas X e Y, pero no la Z */
42   glScalef(escala,escala,1);
43
44   relacAspecto = (float)ventanaAncho/ventanaAlto;
45
46   /* Matriz de perspectiva, utilizando los planos de recorte de
47      cercanía y lejania. */
48   gluPerspective(fovy,relacAspecto,zCerca,zLejos);
49
50   /* Tasladamos la camara en el eje Z (apuntando al origen) */
51   glTranslatef(0,0,-distCamMundo);
52
```

```
53   /* Movemos la camara al centro de la caja.*/
54   glTranslatef(-(maxx+minx)/2,-(maxy+miny)/2,-(maxz+minz)/2);
55  }
56
57  /*****************************************************************
58   *     Funcion: calculaVistaModelo
59   * Descripcion: Situa el modelo en el "centro" del mundo,
60   *           rotandolo segun haya pinchado el usuario.
61   *    Entradas: Ninguna.
62   *     Salidas: Ninguna.
63   *****************************************************************/
64  void calculaVistaModelo(void)
65  {
66   glMatrixMode(GL_MODELVIEW);
67
68   glLoadIdentity();
69
70   // Rotamos el mundo
71   glTranslatef((maxx+minx)/2,(maxy+miny)/2,(maxz+minz)/2);
72   glRotatef(rotx,1,0,0);
73   glRotatef(roty,0,1,0);
74   glRotatef(rotz,0,0,1);
75   glTranslatef(-(maxx+minx)/2,-(maxy+miny)/2,-(maxz+minz)/2);
76
77  }
78
79  /*****************************************************************
80   *     Funcion: dibuja_hueso
81   * Descripcion: Dibuja el hueso especificado en pantalla.
82   *    Entradas: marcaorigen --> Numero de la marca origen del hueso
83   *              marcafin --> Numero de la marca destino del hueso
84   *              frame --> Numero de frame actual
85   *              radio --> Radio del cilindro a dibujar
86   *     Salidas: Ninguna.
87   *****************************************************************/
88  void dibuja_hueso(int marcaorigen, int marcafin, int frame, float radio)
89  {
90    GSList *Nodo;
91    TPunto3D *punto3D;
92    GLUquadricObj *qobj;
93    double OrigenX, OrigenY, OrigenZ;
94    double DestinoX, DestinoY, DestinoZ;
95    double EjeX, EjeY, EjeZ;
96    double EjeRotX, EjeRotY, EjeRotZ;
97    double EjeObjetivoX=0, EjeObjetivoY=0, EjeObjetivoZ=1;
98    float Altura;
99    float Angulo;
100
101  Nodo = (g_slist_nth ( modelo3D[marcaorigen].lista, frame));
102
103  if (Nodo != NULL){
104     punto3D = Nodo->data;
105   OrigenX = punto3D->x;
106   OrigenY = punto3D->y;
107   OrigenZ = punto3D->z;
108
109   Nodo = (g_slist_nth ( modelo3D[marcafin].lista, frame));
110
111   punto3D = Nodo->data;
112   DestinoX = punto3D->x;
```

```
113     DestinoY = punto3D->y;
114     DestinoZ = punto3D->z;
115
116     EjeX = DestinoX - OrigenX;
117     EjeY = DestinoY - OrigenY;
118     EjeZ = DestinoZ - OrigenZ;
119
120     /* Altura del cilindro a dibujar */
121     Altura = (float) sqrt(EjeX*EjeX + EjeY*EjeY + EjeZ*EjeZ);
122
123     glMatrixMode(GL_MODELVIEW);
124     glPushMatrix();
125
126     qobj = gluNewQuadric ();
127     gluQuadricDrawStyle (qobj, GLU_FILL);
128
129     /* Primero trasladamos para que el origen de las coordenadas del mundo
130      coincidan con el punto origen del cilindro */
131     glTranslatef(OrigenX, OrigenY, OrigenZ);
132
133     /* Angulo entre los vectores del eje deseado del cilindro y el eje Z */
134     Angulo = acos((EjeX*EjeObjetivoX + EjeY*EjeObjetivoY
135        + EjeZ*EjeObjetivoZ)/Altura)*360/(2*MI_PI);
136
137     /* Calculamos el eje de rotacion  */
138     EjeRotX = EjeObjetivoY*EjeZ - EjeY*EjeObjetivoZ;
139     EjeRotY = EjeObjetivoZ*EjeX - EjeZ*EjeObjetivoX;
140     EjeRotZ = EjeObjetivoX*EjeY - EjeX*EjeObjetivoY;
141
142     glRotatef(Angulo,EjeRotX,EjeRotY,EjeRotZ); /* Y rotamos. :) */
143
144     /* Si es una marca perdida, no la dibujamos */
145     if ((OrigenX < 999998.) && (DestinoX < 999998.)) {
146
147      /* Dibujamos el cilindro */
148      glDisable(GL_LIGHTING);
149      gluQuadricDrawStyle (qobj, GLU_FILL);
150      gluCylinder(qobj,radio,radio,Altura,5,5);
151      glEnable(GL_LIGHTING);
152
153      /* Dibujamos las "tapas" */
154      gluDisk(qobj,0,radio,10,10);
155      glTranslatef(0,0,Altura);
156      gluDisk(qobj,0,radio,10,10);
157     }
158    }
159
160    glPopMatrix();
161   }
162
163   /*************************************************************
164    *     Funcion: dibujaMarcas
165    * Descripcion: Dibuja la marca especificada en pantalla.
166    *    Entradas: marca --> Numero de la marca a pintar
167    *              tamano --> Radio de la esfera a dibujar
168    *              frame --> Frame actual
169    *     Salidas: Ninguna.
170    *************************************************************/
171   void dibujaMarcas(int marca, double tamano, int frame)
172   {
```

```
173   GSList *Nodo;
174   TPunto3D *punto3D;
175   GLUquadricObj *qobj;
176
177   glMatrixMode(GL_MODELVIEW);
178   glPushMatrix();
179
180   if(posicion3D == VISTA_BRAZOS)
181   {
182    Nodo = (g_slist_nth ( modelo3D[marca].lista, frame));
183
184    if (Nodo != NULL){
185     punto3D = Nodo->data;
186     glTranslatef(punto3D->x,punto3D->y,punto3D->z);
187
188     qobj = gluNewQuadric ();
189
190     /* Dibujamos el esfera */
191     glDisable(GL_LIGHTING); //Comentar si queremos detallado
192     gluQuadricDrawStyle (qobj, GLU_FILL);
193     gluSphere (qobj, tamano, 10, 10);
194     glEnable(GL_LIGHTING); //Comentar si queremos detallado
195    }
196   }
197   else if(posicion3D == VISTA_CARA)
198   {
199    Nodo = (g_slist_nth ( modelo3DGestos[marca].gestos_lista, frame));
200
201    if (Nodo != NULL){
202     punto3D = Nodo->data;
203     glTranslatef(punto3D->x,punto3D->y,punto3D->z);
204
205     qobj = gluNewQuadric ();
206
207     /* Dibujamos el esfera */
208     glDisable(GL_LIGHTING); //Comentar si queremos detallado
209     gluQuadricDrawStyle (qobj, GLU_FILL);
210     gluSphere (qobj, tamano, 10, 10);
211     glEnable(GL_LIGHTING); //Comentar si queremos detallado
212    }
213   }
214
215    glMatrixMode(GL_MODELVIEW);
216    glPopMatrix();
217  }
218
219  /*****************************************************************
220   *     Funcion: renderFrame
221   * Descripcion: Calcula y dibuja en pantalla el frame actual
222   *    Entradas: Ninguna.
223   *     Salidas: Ninguna.
224   *****************************************************************/
225  void renderFrame(void)
226  {
227
228   GtkWidget *hscale_frame;
229
230   double tamano_marca = 8;
231   double tamano_huesos = 2;
232
```

```
233   int i;
234   int frameactual;
235
236   if (ficheroMARCAS!=NULL) {
237
238    /* Localizamos los widgets que vamos a usar... */
239    hscale_frame = glade_xml_get_widget (gxml, "hscale_frame");
240    frameactual = (int) gtk_range_get_value (GTK_RANGE (hscale_frame));
241
242    glClearColor( .0f, .0f, .0f, 1.0f );
243
244    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
245
246    /* Ajustamos la matriz del mundo */
247    calculaVistaModelo();
248
249    // Pintamos la caja
250    dibujaCaja();
251
252
253    if(posicion3D == VISTA_BRAZOS)
254    {
255     // Pintamos las marcas
256     glEnable(GL_LIGHTING);
257     for (i=0; i<(InfoFichero.num_canales/3); i++) {
258       glColor3ub(colores[marcas[i].color].r,
259          colores[marcas[i].color].g,
260         colores[marcas[i].color].b);
261         dibujaMarcas(i, tamano_marca, frameactual);
262      }
263
264     // Dibujamos el esqueleto
265     glEnable(GL_LIGHTING);
266     for (i=0; i<numero_huesos; i++) {
267      glColor3ub(colores[huesos[i].color].r,
268          colores[huesos[i].color].g,
269        colores[huesos[i].color].b);
270      dibuja_hueso(huesos[i].origen, huesos[i].destino,
271           frameactual, tamano_huesos);
272     }
273    }
274    else if(posicion3D == VISTA_CARA)
275    {
276     tamano_marca = 4;
277     // Pintamos las marcas de la cara
278     glEnable(GL_LIGHTING);
279     for (i=0; i<(InfoFicheroGestos.gestos_num_canales/3); i++) {
280      //~ if (marcas[i].visible) {
281        glColor3ub(colores[marcas_cara[i].color].r,
282         colores[marcas_cara[i].color].g,
283        colores[marcas_cara[i].color].b);
284        dibujaMarcas(i, tamano_marca, frameactual);
285      //~ }
286     }
287    }
288   } // Fin del if ficheroMARCAS!=NULL
289   else {
290    glClearColor( .85f, .85f, .85f, 1.0f );//Color gris
291
292       glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

```
293   }
294  }
295
296  /*****************************************************************
297   *     Funcion: iniciar_gl
298   * Descripcion: Inicializa los parametros relacionados con el widget
299   *              de OpenGL
300   *    Entradas: widget --> Puntero al widget GtkGlExt
301   *              data --> Puntero con los datos de los como parametros
302   *     Salidas: Ninguna.
303   *****************************************************************/
304  void iniciar_gl (GtkWidget *widget, gpointer data) {
305
306   caracter = 0;
307   GdkGLContext *glcontext = gtk_widget_get_gl_context (widget);
308   GdkGLDrawable *gldrawable = gtk_widget_get_gl_drawable (widget);
309
310   static GLfloat light0_ambient[] =  {0.2f, 0.2f, 0.2f, 1.0f};
311   static GLfloat light0_diffuse[] =  {.8f, .8f, 0.8f, 1.0f};
312   static GLfloat light0_position[] = {1.0f, 1.0f, 1.0f, 0.0f};
313
314   /* Comienza la parte de OpenGL... */
315   if (!gdk_gl_drawable_gl_begin (gldrawable, glcontext)){
316    g_print ("*** ERROR: Problem with init GL\n");
317    return;
318   }
319
320   glEnable(GL_LIGHTING);
321   glEnable(GL_LIGHT0);
322
323   glLightfv(GL_LIGHT0, GL_AMBIENT, light0_ambient);
324   glLightfv(GL_LIGHT0, GL_DIFFUSE, light0_diffuse);
325   glLightfv(GL_LIGHT0, GL_POSITION, light0_position);
326
327   /* Activamos el zbuffer */
328   glEnable(GL_DEPTH_TEST);
329   glDepthFunc(GL_LEQUAL);  /* Tipo de test de profundidad */
330   glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
331
332   glEnable(GL_COLOR_MATERIAL);
333   glColorMaterial (GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
334
335   gdk_gl_drawable_gl_end (gldrawable);
336  }
337
338  /*****************************************************************
339   *     Funcion: reescalar_gl
340   * Descripcion: Manejador del evento de reescalado del Widget
341   *    Entradas: widget --> Puntero al widget GtkGlExt
342   *              event --> Puntero con caracteristicas del evento
343   *              data --> Puntero con los datos de los parametros
344   *     Salidas: Devuelve TRUE si no se produjo error.
345   *****************************************************************/
346  gboolean reescalar_gl (GtkWidget *widget, GdkEventConfigure *event,
347            gpointer data)
348  {
349    GdkGLContext *glcontext = gtk_widget_get_gl_context (widget);
350    GdkGLDrawable *gldrawable = gtk_widget_get_gl_drawable (widget);
351
352    gdk_gl_drawable_gl_begin (gldrawable, glcontext);
```

```
353
354     ventanaAncho = widget->allocation.width;
355     ventanaAlto = widget->allocation.height;
356
357     calculaProyeccion();
358     renderFrame();
359
360     gdk_gl_drawable_gl_end (gldrawable);
361
362     return TRUE;
363   }
364
365   /*****************************************************************
366    *     Funcion: dibujar_gl
367    * Descripcion: Manejador del evento de dibujado del Widget
368    *    Entradas: widget --> Puntero al widget GtkGlExt
369    *              event --> Puntero con caracteristicas del evento
370    *              data --> Puntero con datos de los parametros
371    *     Salidas: Devuelve TRUE si no se produjo error.
372    *****************************************************************/
373   gboolean dibujar_gl (GtkWidget *widget, GdkEventExpose *event,
374          gpointer data) {
375
376    GdkGLContext *glcontext = gtk_widget_get_gl_context (widget);
377    GdkGLDrawable *gldrawable = gtk_widget_get_gl_drawable (widget);
378
379    GtkWidget *hscale_frame;
380    GtkWidget *boton_play;
381    GtkWidget *spin_ver_velocidad;
382
383    int nuevoframe, frameactual;
384    int velocidad, i;
385
386
387    if (!gdk_gl_drawable_gl_begin (gldrawable, glcontext)){
388     g_print ("*** ERROR: Problem in dibujar\n");
389     return FALSE;
390    }
391
392    /* Comenzamos la parte de OpenGL */
393    ventanaAncho = widget->allocation.width;
394    ventanaAlto = widget->allocation.height;
395
396
397    hscale_frame = glade_xml_get_widget (gxml, "hscale_frame");
398    frameactual = (int) gtk_range_get_value (GTK_RANGE (hscale_frame));
399
400    calculaProyeccion();
401    renderFrame();
402
403    if (frameactual == 1 && frameInitialized == 0)
404    {
405     initializeFaceCapture();
406     showImages();
407    }
408
409
410    /* Enviamos la informacion y terminamos */
411    if (gdk_gl_drawable_is_double_buffered (gldrawable)) {
412     gdk_gl_drawable_swap_buffers (gldrawable);
```

```
413   }
414   else {
415    glFlush ();
416   }
417
418   gdk_gl_drawable_gl_end (gldrawable);
419
420   /* Importantisimo! Permitimos dibujarse al resto de widgets */
421   while (g_main_iteration(FALSE));
422
423    /* Antes de salir vemos si esta el boton del play pulsado...
424      Si esta pulsado, realimentamos */
425
426   boton_play = glade_xml_get_widget (gxml, "boton_play");
427   spin_ver_velocidad = glade_xml_get_widget (gxml, "spin_ver_velocidad");
428     velocidad = (int)
429    gtk_spin_button_get_value (GTK_SPIN_BUTTON (spin_ver_velocidad));
430
431   if (gtk_toggle_button_get_active (GTK_TOGGLE_BUTTON(boton_play))) {
432
433       if (frameactual < (InfoFichero.num_frames - 1)){
434     if (velocidad >= 0) {
435              nuevoframe = frameactual + 1 + velocidad;
436      if (nuevoframe > InfoFichero.num_frames-1) {
437       nuevoframe = InfoFichero.num_frames-1;
438      }
439           }
440           else {  /* Tenemos retardo */
441              for (i=0; i>velocidad*2; i--) {
442                /* Permitimos que se dibujen otros widgets
443         mientras no hacemos nada */
444                while (g_main_iteration(FALSE));
445              }
446              nuevoframe = frameactual + 1;
447           }
448       }
449       else {
450     nuevoframe = 0;
451    }
452    if(velocidad > 0) {
453     usleep(10000 / (velocidad + 1));
454
455    }
456    else if(velocidad < 0){
457     usleep((-velocidad + 1) * 10000);
458    }
459    else {
460     usleep(10000);
461    }
462
463       gtk_range_set_value (GTK_RANGE (hscale_frame), nuevoframe);
464    }
465    return TRUE;
466  }
467
468  /*****************************************************************
469   *     Funcion: getImagePixbufOpencvExample
470   * Descripcion: Obtiene una imagen, la escala y la devuelve para
471   *              representarla en un GTKImage.
472   *    Entradas: filename --> Puntero al nombre del fichero a abrir.
```

```
473   *     Salidas: puntero a GdkPixbuf.
474   ************************************************************/
475  GdkPixbuf* getImagePixbufOpencvExample(const char* filename){
476
477   IplImage *imgPix = cvLoadImage(filename, CV_LOAD_IMAGE_COLOR);
478   gchar *archivoVideo = g_strdup(filename);
479
480   GdkPixbuf *pix_src, *pix_dst;
481      pix_src = gdk_pixbuf_new_from_data((guchar*)imgPix->imageData,
482          GDK_COLORSPACE_RGB, FALSE,
483                              imgPix->depth, imgPix->width,
484            imgPix->height, imgPix->widthStep,
485          NULL, NULL);
486
487   archivoVideo = strrchr(archivoVideo,'.');
488
489   if(strcmp(archivoVideo,".png") != 0) {
490    pix_dst = gdk_pixbuf_scale_simple(pix_src, imgPix->width/10,
491          imgPix->height/10, GDK_INTERP_BILINEAR);
492   }
493   else {
494    pix_dst = gdk_pixbuf_scale_simple(pix_src, imgPix->width,
495           imgPix->height, GDK_INTERP_BILINEAR);
496   }
497      cvReleaseImage( &imgPix );
498
499      return pix_dst;
500  }
501
502  /*********************************************************************
503   *     Funcion: motion_notify_event
504   * Descripcion: Manejador del evento de arrastrar el raton en el Widget
505   *    Entradas: widget --> Puntero al widget GtkGlExt
506   *              event --> Puntero con caracteristicas del evento ocurrido
507   *    data --> Puntero con datos que se le pasan como parametros
508   *     Salidas: Devuelve TRUE si no se produjo error.
509   *********************************************************************/
510  gboolean motion_notify_event (GtkWidget *widget, GdkEventButton *event,
511           gpointer data)
512  {
513    int x, y, dx, dy;
514    GdkModifierType state = 0;
515    GtkWidget *boton_play;
516
517    x = event->x;
518    y = event->y;
519
520    dx = x - inicioX;
521    dy = y - inicioY;
522    state = event->state;
523
524    /* Rotamos con el arrastre del boton izquierdo... */
525    if (state & GDK_BUTTON1_MASK) {  /* Boton izquierdo pulsado */
526      rotx = getModulo(inicioRotX + factorRotacion*dy,360);
527      rotz = getModulo(inicioRotZ + factorRotacion*dx,360);
528      calculaVistaModelo();
529
530    } else if (state && GDK_BUTTON2_MASK) {  /* Boton central pulsado */
531      trans[0] = inicioTransX- factorTrans*dx;
532      trans[1] = inicioTransY+ factorTrans*dy;
```

```
533       calculaVistaModelo();
534       //~ calculaProyeccion();
535     }
536
537     boton_play = glade_xml_get_widget(gxml,"boton_play");
538
539     /* Si el boton del play no esta activo, redibujamos
540       (si esta activo, el redibujado es automatico). */
541     if ((gtk_toggle_button_get_active (GTK_TOGGLE_BUTTON(boton_play))
542               ==FALSE) && (state)) {
543       dibujar_gl (widget, NULL, NULL);
544     }
545
546    return TRUE;
547  }
548
549  /*****************************************************************
550   *     Funcion: button_press_event
551   * Descripcion: Manejador del evento de pinchar con el raton en el Widget
552   *    Entradas: widget --> Puntero al widget GtkGlExt
553   *              event --> Puntero con caracteristicas del evento ocurrido
554   *    data --> Puntero con datos que se le pasan como parametros
555   *     Salidas: Devuelve TRUE si no se produjo error.
556   *****************************************************************/
557  gboolean button_press_event (GtkWidget *widget, GdkEventButton *event,
558          gpointer data)
559  {
560     inicioRotX = rotx;
561     inicioRotY = roty;
562     inicioRotZ = rotz;
563
564     inicioX = event->x;
565     inicioY = event->y;
566
567     inicioTransX = trans[0];
568     inicioTransY = trans[1];
569
570     return TRUE;
571  }
572
573  /*****************************************************************
574   *     Funcion: scroll_event
575   * Descripcion: Manejador del evento de rueda del raton en el Widget
576   *    Entradas: widget --> Puntero al widget GtkGlExt
577   *              event --> Puntero con caracteristicas del evento ocurrido
578   *    data --> Puntero con datos que se le pasan como parametros
579   *     Salidas: Devuelve TRUE si no se produjo error.
580   *****************************************************************/
581  gboolean scroll_event (GtkRange *range, GdkEventScroll *event,
582         gpointer data){
583     if (event->direction == GDK_SCROLL_DOWN)
584        escala-=0.01;
585     if (event->direction == GDK_SCROLL_UP)
586        escala+=0.01;
587     gtk_widget_queue_draw(GTK_WIDGET((GtkWidget *)data));
588
589     return TRUE;
590  }
591
592  /*****************************************************************
```

```
593   *     Funcion: key_press_event
594   * Descripcion: Manejador de eventos de teclado en la Interfaz.
595   *    Entradas: widget --> Puntero al widget GtkGlExt
596   *              event --> Puntero con caracteristicas del evento ocurrido
597   *    data --> Puntero con datos que se le pasan como parametros
598   *     Salidas: Devuelve TRUE si no se produjo error.
599   ****************************************************************/
600  gboolean key_press_event (GtkWidget *widget, GdkEventKey *event,
601         gpointer data){
602
603   GtkWidget *sec_video_frames;
604   GtkWidget *notebook_edicion;
605   GtkWidget *show_tools_panel;
606   GtkWidget *show_preprocess_video;
607
608   show_tools_panel = glade_xml_get_widget(gxml, "show_tools_panel");
609   show_preprocess_video = glade_xml_get_widget(gxml,
610            "show_preprocess_video");
611
612   sec_video_frames = glade_xml_get_widget (gxml, "sec_video_frames");
613   notebook_edicion = glade_xml_get_widget (gxml, "notebook_edicion");
614
615   switch (event->keyval) {
616    case GDK_plus:
617     escala += 0.01;
618     calculaProyeccion();
619    break;
620    case GDK_minus:
621     escala -= 0.01;
622     calculaProyeccion();
623    break;
624    case GDK_F1:
625     if(GTK_WIDGET_VISIBLE (sec_video_frames)) {
626      //~ gtk_widget_hide(sec_video_frames);
627      gtk_check_menu_item_set_active(show_preprocess_video, FALSE);
628     }else {
629      //~ gtk_widget_show(sec_video_frames);
630      gtk_check_menu_item_set_active(show_preprocess_video, TRUE);
631     }
632     break;
633    case GDK_F2:
634     if(GTK_WIDGET_VISIBLE (notebook_edicion)) {
635      //~ gtk_widget_hide(notebook_edicion);
636      gtk_check_menu_item_set_active(show_tools_panel, FALSE);
637     }else {
638      //~ gtk_widget_show(notebook_edicion);
639      gtk_check_menu_item_set_active(show_tools_panel, TRUE);
640     }
641     break;
642    case GDK_F3:
643     break;
644    default:
645     return FALSE;
646   }
647
648   gtk_widget_queue_draw(GTK_WIDGET((GtkWidget *)data));
649
650   return TRUE;
651  }
652
```

```
653 /*********************************************************************
654  *     Funcion: abrir_fichero_bon
655  * Descripcion: Realiza la apertura del fichero de configuración de las
656  *          marcas y huesos de los brazos, y los carga en su estructura
657  *          de datos marcas y huesos.
658  *     Entradas: fichero --> Puntero al nombre del fichero a abrir.
659  *     Salidas: Devuelve -1 si hubo algun error. 0 en caso contrario.
660  *********************************************************************/
661 int abrir_fichero_bon (char *nombre)
662 {
663   FILE *ficheroBON;
664   int num_marcas, num_huesos;
665   int i;
666
667   if ((ficheroBON = fopen (nombre, "rw"))==NULL) return 1;
668   else {
669
670     fscanf (ficheroBON, "%d\t%d", &num_marcas, &num_huesos);
671     if (num_marcas != (InfoFichero.num_canales/3)){
672       mostrar_popup ("El fichero de Esqueleto NO es valido.\n
673     Causa: Diferente numero de canales",
674      GTK_MESSAGE_ERROR, GTK_BUTTONS_CLOSE);
675       return 1;
676     };
677
678     for (i=0; i<num_marcas; i++)
679       fscanf (ficheroBON, "%d\t%d", &marcas[i].color, &marcas[i].visible);
680
681     for (i=0; i<num_huesos; i++)
682       fscanf (ficheroBON, "%d\t%d\t%d", &huesos[i].origen,
683      &huesos[i].destino, &huesos[i].color);
684
685     numero_huesos = num_huesos; // Actualizamos la variable
686
687     fclose (ficheroBON);
688     return 0;
689   }
690 }
691
692 /*********************************************************************
693  *     Funcion: guardar_fichero
694  * Descripcion: Guarda los datos que tenemos en memoria en un
695  *            fichero MARCAS
696  *    Entradas: nombre --> Puntero al nombre del fichero a guardar.
697  *     Salidas: Devuelve 1 si hubo algun error. 0 en caso contrario.
698  *********************************************************************/
699 int guardar_fichero (char *nombre)
700 {
701   int i,j, longitud;
702   GSList *Nodo;
703   TPunto3D *punto3D;
704   char cadaux [500];
705   FILE *ficheroSAVE=NULL;
706
707   if ((ficheroSAVE = fopen (nombre, "w"))==NULL) return 1;
708
709   /* Parte de DATA_SECTION*/
710
711   cadaux[0] = 'X';
712   rewind(ficheroMARCAS);
```

```
713     fgets (cadaux, 499, ficheroMARCAS);
714     longitud = strlen (cadaux);
715     while (cadaux[0]!='0') {
716       fputs (cadaux, ficheroSAVE);
717       fgets (cadaux, 499, ficheroMARCAS);
718     }
719
720     /* Se rellenan los valores "interesantes" de nuestras listas */
721    printf("Guardando poses de brazos...\n");
722
723     for (j=0; j<InfoFichero.num_frames; j++) {
724       for (i=0; i<InfoFichero.num_canales/3; i++) {
725         Nodo = (g_slist_nth ( modelo3D[i].lista, j));
726
727     if (Nodo != NULL)
728     {
729      punto3D = Nodo->data;
730      fprintf (ficheroSAVE, "%i \t %i \t %f\r\n", i*3, j, punto3D->x);
731      fprintf (ficheroSAVE, "%i \t %i \t %f\r\n", i*3+1, j, punto3D->y);
732      fprintf (ficheroSAVE, "%i \t %i \t %f\r\n", i*3+2, j, punto3D->z);
733     }
734        }
735     }
736     fclose (ficheroSAVE);
737     printf("Fichero brazos guardado correctamente\n");
738     return 0;
739
740  }
741
742  /*********************************************************************
743   *     Funcion: showImages
744   * Descripcion: Muestra las imágenes en los widgets
745   *    Entradas: Ninguna.
746   *     Salidas: Devuelve 1 si error.
747   *********************************************************************/
748  int showImages(void)
749  {
750   GtkWidget *hscale_frame;
751
752   int i;
753   int frameactual;
754
755   DIR *dip;
756      struct dirent *dit;
757   char name [50]= DIR_IMAGES;
758
759   char nameImg [50];
760
761   /* Localizamos los widgets que vamos a usar... */
762   hscale_frame = glade_xml_get_widget (gxml, "hscale_frame");
763   frameactual = (int) gtk_range_get_value (GTK_RANGE (hscale_frame));
764
765   if (ficheroMARCAS!=NULL) {
766    if (frameactual == 1 && archivoImagenVideo == NULL)
767    {
768     archivoImagenVideo = strrchr(ficheroAVIAbierto,'/');
769     frameInitialized = 1;
770
771     archivoImagenVideo[strlen(archivoImagenVideo) - 4] = '\0';
772
```

```
773     if ((dip = opendir(name)) == NULL){
774      perror("opendir");
775      return 0;
776     }
777    }
778
779    strcat(name, archivoImagenVideo);
780
781    char str [10];
782    sprintf (str,"%04i", frameactual);
783
784    strcpy(nameImg,name);
785    strcat(nameImg,archivoImagenVideo);
786    strcat(nameImg,str);
787    strcat(nameImg,".jpg");
788
789    /* load the image*/
790    img = cvLoadImage( nameImg, CV_LOAD_IMAGE_COLOR );
791
792    /* always check */
793    if( img == 0 ) {
794     fprintf( stderr, "No se puede cargar el fichero %s!\n", nameImg );
795     return 1;
796    }
797    else {
798     createAdquisitionVideo(drawing_area_1);
799
800     createAdquisitionVideo2(drawing_area_2);
801
802     pre_procesing_images(img);
803
804     /Para que en el frame 0 se pueda redibujar_gl ()*/
805     if (frameactual != 1) {
806      cvReleaseImage( &img );
807      cvReleaseImage( &imageOF );
808     }
809
810     createAdquisitionVideo3(drawing_area_3);
811
812     createAdquisitionVideo4(drawing_area_4);
813
814     createAdquisitionVideo5(drawing_area_5);
815
816     /*Para que en el frame 0 se pueda redibujar_gl ()*/
817     if (frameactual != 1)
818      free_memory();
819
820     return 0;
821    }
822
823   } // Fin del if ficheroMARCAS!=NULL
824  }
825
826  /*********************************************************************
827   *     Funcion: createAdquisitionVideo
828   * Descripcion: Función que crea una textura en OpenGl a partir de
829   *              un video en OpenCv
830   *    Entradas: GtkWidget *: Widget en el que se renderiza la imagen.
831   *     Salidas: Devuelve TRUE si no se produjo error.
832   *********************************************************************/
```

```
833 gboolean createAdquisitionVideo(GtkWidget *widget){
834
835  GdkGLContext *glcontext;
836  GdkGLDrawable *gldrawable;
837
838  if (img != NULL) {
839  glcontext = gtk_widget_get_gl_context (widget);
840  gldrawable = gtk_widget_get_gl_drawable (widget);
841
842
843   if( !imageAdquisition) {
844    imageAdquisition = cvCreateImage( cvSize(img->width,
845               img->height),
846             IPL_DEPTH_8U, img->nChannels );
847   }
848
849   cvResize(img, imageAdquisition, CV_INTER_LINEAR );
850   cvFlip( imageAdquisition, imageAdquisition, 0 );
851
852   if (!gdk_gl_drawable_gl_begin (gldrawable, glcontext)){
853    g_print ("*** ERROR: Problem in dibujar\n");
854   }
855
856   glClear( GL_COLOR_BUFFER_BIT );
857   glLoadIdentity();
858   glTranslatef(0.0f, 0.0f,-1.0f);
859
860   GLenum format = IsBGR(imageAdquisition->channelSeq)
861       ? GL_BGR_EXT : GL_RGBA;
862
863   if (!init) {
864    glGenTextures(1, &imageID1);
865    glBindTexture(GL_TEXTURE_2D, imageID1);
866    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
867    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
868    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
869    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, imageAdquisition->width,
870        imageAdquisition->height, 0,
871        format, GL_UNSIGNED_BYTE, imageAdquisition->imageData);
872    init= 1; // init is over
873   }
874   else {
875    glBindTexture(GL_TEXTURE_2D, imageID1);
876    glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, imageAdquisition->width,
877        imageAdquisition->height, format, GL_UNSIGNED_BYTE,
878        imageAdquisition->imageData);
879   }
880   glPushMatrix();
881   glEnable(GL_TEXTURE_2D);
882
883   glBegin(GL_QUADS);{
884    glTexCoord2f(0, 0); glVertex3f(-1, -1, 0);
885    glTexCoord2f(0, 1); glVertex3f(-1, 1, 0);
886    glTexCoord2f(1, 1); glVertex3f(1, 1, 0);
887    glTexCoord2f(1, 0); glVertex3f(1, -1, 0);
888
889   }glEnd();
890   glPopMatrix();
891
892  if (gdk_gl_drawable_is_double_buffered (gldrawable)){
```

```
893     gdk_gl_drawable_swap_buffers (gldrawable);
894    }
895    else{
896     glFlush ();
897    }
898
899    gdk_gl_drawable_gl_end (gldrawable);
900
901    while (g_main_iteration(FALSE));
902    }
903
904    return TRUE;
905
906   }
```

Funciones más relevantes de *openCVFunctions.c*

```
1   /*********************************************************************
2    *     Funcion: init_video_process
3    * Descripcion: Procesa el fichero AVI, mediante algoritmo de Lucas-Kanade
4    *              se crean los puntos 2D y 3D y se guardan en un fichero .mf
5    *    Entradas: fichero --> Puntero al nombre del fichero a abrir.
6    *     Salidas: Devuelve -1 si hubo algun error. 0 en caso contrario.
7    *********************************************************************/
8   int init_video_process(gchar *ficheroAux) {
9
10   CvCapture *capture;
11   IplImage *frame;
12   numFrame = 0;
13
14   int c;
15
16   gchar *fichero = g_strdup (ficheroAux);
17   free_memory();
18
19   /* Si el archivo .mf  y .gff existe no hacemos nada. */
20   if (write_mf_file(fichero) == 0
21    && write_face_gestures_file(fichero) == 0) {
22
23    /* memoria necesaria para la captura */
24    storage = cvCreateMemStorage(0);
25
26    capture = cvCaptureFromAVI(fichero);
27
28    initializeFaceCapture();
29    if(!capture){
30     return -1;
31    }
32    for(;;) {
33
34     if(!cvGrabFrame(capture)){
35      break;
36     }
37     frame = cvRetrieveFrame(capture);
38     if(!frame){
39      break;
40     }
41     else {
42
```

```
43      /* Hacemos tantas copias de la imagen como
44      procesamientos diferentes queramos hacer. */
45      if( !frame_copy) {
46       frame_copy =
47       cvCreateImage( cvSize(frame->width,frame->height),
48          IPL_DEPTH_8U, frame->nChannels );
49      }
50
51      if( !imageAdquisition) {
52       imageAdquisition =
53       cvCreateImage( cvSize(frame->width,frame->height),
54             IPL_DEPTH_8U, frame->nChannels );
55      }
56
57      if( frame->origin == IPL_ORIGIN_TL ) {
58       cvFlip( frame, frame_copy, 0 );
59       cvFlip( frame, imageAdquisition, 0 );
60      }
61      else {
62       cvFlip( frame, frame_copy, 0 );
63      }
64
65      //Se aplica el algoritmo de L-K mejorado.
66      calcularOpticalFlow(frame);
67      c = cvWaitKey(25);
68
69     }//fin else frame
70    }//fin for
71
72    cvReleaseCapture(&capture);
73    cvReleaseImage(&frame_copy);
74    cvReleaseMemStorage(&storage);
75    caraInicialized = 0;
76       free_memory();
77
78   }
79   else {
80    free_memory();
81   }
82
83   if(ficheroMARCAS != NULL)
84    fclose(ficheroMARCAS);
85   if(ficheroGFACIALES != NULL)
86    fclose(ficheroGFACIALES);
87
88   return 0;
89  }
90
91  /*********************************************************************
92   *     Funcion: calcularOpticalFlow
93   * Descripcion: Implementación del algoritmo de Lucas-Kanade.
94   *    Entradas: frame_optical_flow --> Puntero a Imagen a analizar.
95   *     Salidas: nada.
96   *********************************************************************/
97  void calcularOpticalFlow(IplImage *frame_optical_flow){
98
99   int i, k;// c;
100
101  if( !imageOF )
102  {
```

```
103     cvReleaseImage( &grey );
104     cvReleaseImage( &prev_grey );
105     cvReleaseImage( &pyramid );
106     cvReleaseImage( &prev_pyramid );
107
108     imageOF = cvCreateImage( cvSize( frame_optical_flow->width,
109             frame_optical_flow->height), 8, 3);
110
111     grey = cvCreateImage( cvGetSize(frame_optical_flow), 8, 1 );
112     prev_grey = cvCreateImage( cvGetSize(frame_optical_flow), 8, 1 );
113     pyramid = cvCreateImage( cvGetSize(frame_optical_flow), 8, 1 );
114     prev_pyramid = cvCreateImage( cvGetSize(frame_optical_flow), 8, 1 );
115     points[0] = (CvPoint2D32f*)cvAlloc(MAX_COUNT*sizeof(points[0][0]));
116     points[1] = (CvPoint2D32f*)cvAlloc(MAX_COUNT*sizeof(points[0][0]));
117
118     points3d [0]= (CvPoint3D32f*)cvAlloc(MAX_COUNT*sizeof(points3d[0][0]));
119     points3d [1]= (CvPoint3D32f*)cvAlloc(MAX_COUNT*sizeof(points3d[0][0]));
120
121     status = (char*)cvAlloc(MAX_COUNT);
122     flags = 0;
123    }
124
125
126    cvCopy(frame_optical_flow, imageOF, NULL);
127
128    cvCvtColor(imageOF, grey, CV_BGR2GRAY);
129
130
131    if(need_to_init)
132    {
133     /* automatic initialization */
134     IplImage* eig = cvCreateImage( cvGetSize(grey), 32, 1 );
135     IplImage* temp = cvCreateImage( cvGetSize(grey), 32, 1 );
136
137     /* Se inicializan los puntos de seguimiento para los huesos */
138     initializeCoordinates(1);
139
140
141     cvFindCornerSubPix( grey, points[1], count,
142          cvSize(win_size,win_size), cvSize(-1,-1),
143         cvTermCriteria(CV_TERMCRIT_ITER|CV_TERMCRIT_EPS,20,0.03));
144     cvReleaseImage( &eig );
145     cvReleaseImage( &temp );
146
147     add_remove_pt = 0;
148    }
149    else if( count > 0 )
150    {
151     cvCalcOpticalFlowPyrLK( prev_grey, grey, prev_pyramid, pyramid,
152       points[0], points[1], count,
153       cvSize(win_size,win_size), 3, status, 0,
154       cvTermCriteria(CV_TERMCRIT_ITER|CV_TERMCRIT_EPS,20,0.03),
155       flags );
156     flags |= CV_LKFLOW_PYR_A_READY;
157
158     for( i = k = 0; i < count; i++ )
159     {
160      if( add_remove_pt )
161      {
162       double dx = pt.x - points[1][i].x;
```

```
163       double dy = pt.y - points[1][i].y;
164
165       if( dx*dx + dy*dy <= 25 )
166       {
167        add_remove_pt = 0;
168        continue;
169       }
170      }
171
172      if( !status[i] )
173       continue;
174
175      points[1][k++] = points[1][i];
176
177      [...]
178
179      /* Cada uno de los puntos */
180      cvCircle( imageOF, cvPointFrom32f(points[1][i]), 3,
181          CV_RGB(0,255,0), -1, 8,0);
182
183      /* Vectores de movimiento */
184      cvLine(imageOF,cvPointFrom32f(points[0][i]),
185       cvPointFrom32f(points[1][i]),CV_RGB(255,0,0),1,8,0);
186
187      /* Escribimos info en el fichero de marcas de huesos. */
188      if (i < 6 && ficheroMARCAS != NULL) {
189
190       if (MarcasEscritas == InfoFichero.num_canales) {
191        MarcasEscritas = 0;
192       }
193
194       /* Se ponen signos negativos para renderizar
195       bien en openGL con coords buenas*/
196       fprintf (ficheroMARCAS, "%i \t%i \t%f\r\n",
197         MarcasEscritas++, numFrame - 1, -points[1][i].x);
198       fprintf (ficheroMARCAS, "%i \t%i \t%f\r\n",
199         MarcasEscritas++, numFrame - 1, points3d[1][i].z);
200       fprintf (ficheroMARCAS, "%i \t%i \t%f\r\n",
201         MarcasEscritas++, numFrame - 1, -points[1][i].y);
202        }
203
204      /* Escribimos info en el fichero de marcas de gestos faciales. */
205      if (i>15 && i < 32 && ficheroGFACIALES != NULL) {
206
207       if (MarcasEscritasGestos ==
208        InfoFicheroGestos.gestos_num_canales) {
209        MarcasEscritasGestos = 0;
210       }
211
212       /* Se ponen signos negativos para renderizar bien
213        en openGL con coords buenas*/
214       fprintf (ficheroGFACIALES, "%i \t%i \t%f\r\n",
215         MarcasEscritasGestos++, numFrame - 1, -points[1][i].x);
216       fprintf (ficheroGFACIALES, "%i \t%i \t%f\r\n",
217         MarcasEscritasGestos++, numFrame - 1, points3d[1][i].z);
218       fprintf (ficheroGFACIALES, "%i \t%i \t%f\r\n",
219         MarcasEscritasGestos++, numFrame - 1, -points[1][i].y);
220        }
221
222
```

```
223    }//fin for para recorrer todos los puntos en cada frame
224
225    /* Para no perder puntos */
226    if (k != count) {
227
228     points[1][k].x =  points[0][k].x;
229     points[1][k].y =  points[0][k].y;
230     k ++;
231
232    }
233    count = k;
234
235   }
236
237   /* Para inicializar puntos con clics de ratón */
238   if( add_remove_pt && count < MAX_COUNT )
239   {
240    points[1][count++] = cvPointTo32f(pt);
241    cvFindCornerSubPix( grey, points[1] + count - 1,
242       1,cvSize(win_size,win_size), cvSize(-1,-1),
243      cvTermCriteria(CV_TERMCRIT_ITER|CV_TERMCRIT_EPS,20,0.03));
244    add_remove_pt = 0;
245    if(count == 6 ) {
246     initializeCoordinates(0);
247    }
248   }
249
250   /* Se recalculan las longitudes de los brazos y angulos */
251   longAnteBrIzqdo = distance(points[1][2].x,points[1][2].y,
252    points[1][0].x,points[1][0].y);
253   longAnteBrDrcho = distance(points[1][3].x,points[1][3].y,
254    points[1][1].x,points[1][1].y);
255   longBrazoIzqdo  = distance(points[1][4].x,points[1][4].y,
256    points[1][2].x,points[1][2].y);
257   longBrazoDrcho  = distance(points[1][5].x,points[1][5].y,
258    points[1][3].x,points[1][3].y);
259
260   /* Se copian las caracteristicas de la imagen del frame
261    actual como frame anterior */
262   CV_SWAP( prev_grey, grey, swap_temp );
263   CV_SWAP( prev_pyramid, pyramid, swap_temp );
264   CV_SWAP( points[0], points[1], swap_points );
265   CV_SWAP( points3d[0], points3d[1], swap_points3d);
266
267   need_to_init = 0;
268
269   detect_face_eyes_mouth(imageOF);
270   find_and_draw_contours(imageOF);
271
272  }
```

Funciones más relevantes de *splines.c*

```
1  /*********************************************************************
2   *     Funcion: suavizar
3   * Descripcion: Suaviza la trayectoria del canal pasado como parametro.
4   *    Entradas: desde --> Frame origen del intervalo
5   *              hasta --> Frame destino del intervalo
6   *              canal --> Numero de canal a suavizar
```

```
 7    *              suavizado --> Factor de suavizado a utilizar
 8    *     Salidas: Ninguna.
 9    ************************************************************************/
10   void suavizar (int desde, int hasta, int canal, int suavizado)
11   {
12     int nerrores_intervalo=0;
13     double vaux [MAX_PTOS];
14     int i, j, desde_prop, hasta_prop;
15     TPunto3D *Punto3D;
16     GSList *Nodo;
17
18     /* Veamos si el intervalo tiene algun error */
19     for (i=0; i<hasta-desde+1; i++) {
20       Nodo = (g_slist_nth ( modelo3D[canal].lista, desde+i));
21       Punto3D = Nodo->data;
22       switch (j%3) {
23       case 0 : { if (Punto3D->x > 999998) nerrores_intervalo++; break; }
24       case 1 : { if (Punto3D->y > 999998) nerrores_intervalo++; break; }
25       case 2 : { if (Punto3D->z > 999998) nerrores_intervalo++; break; }
26       };
27     }
28
29     if (nerrores_intervalo != 0)
30       mostrar_popup (g_strdup_printf ("El intervalo seleccionado no puede
31       tener\n errores para proceder al suavizado.\n
32       El intervalo %d - %d del canal [%s] tiene %d errores.",
33       desde, hasta, modelo3D[canal].nombre->str,
34       nerrores_intervalo) , GTK_MESSAGE_ERROR, GTK_BUTTONS_CLOSE);
35     else {  /* El intervalo elegido no tiene errores */
36       /* Creamos el vector de puntos... */
37       for (j=canal; j<(canal+3); j++) {  // Recorremos eje X,Y,Z del canal
38         desde_prop = hasta_prop = desde;
39         while (hasta != hasta_prop) {
40           if (hasta-hasta_prop > MAX_PTOS) hasta_prop += (MAX_PTOS -1);
41           else hasta_prop = hasta;
42
43           /* Introducimos errores cada tantos puntos como indique el */
44           /* factor de suavizado. Luego pasaremos a interpolar...    */
45
46           for (i=0; i<hasta_prop-desde_prop+1; i++) {
47             Nodo = (g_slist_nth ( modelo3D[canal].lista, desde_prop+i));
48             Punto3D = Nodo->data;
49             switch (j%3) {
50               case 0 : { if ((i%suavizado==0) || (i==hasta_prop-desde_prop))
51                          vaux [i] = Punto3D->x; else vaux [i] = 999999;
                              break; }
52               case 1 : { if ((i%suavizado==0) || (i==hasta_prop-desde_prop))
53                          vaux [i] = Punto3D->y; else vaux [i] = 999999;
                              break; }
54               case 2 : { if ((i%suavizado==0) || (i==hasta_prop-desde_prop))
55                          vaux [i] = Punto3D->z; else vaux [i] = 999999;
                              break; }
56             };
57           }
58
59           interpola (hasta_prop-desde_prop+1, vaux);
60
61           /* Guardamos el resultado obtenido en la estructura de datos */
62
63           for (i=0; i<hasta_prop-desde_prop+1; i++) {
```

```
64            Nodo = (g_slist_nth ( modelo3D[canal].lista, desde_prop+i));
65            Punto3D = Nodo->data;
66            switch (j%3) {
67              case 0 : { Punto3D->x = vaux [i]; break; }
68              case 1 : { Punto3D->y = vaux [i]; break; }
69              case 2 : { Punto3D->z = vaux [i]; break; }
70            };
71          }
72          desde_prop = hasta_prop+1;
73        } /* Fin del while */
74      } /* Fin del for con el que recorremos XYZ del canal */
75    } /* Else (si no hubo error) */
76  }
77
78  /*****************************************************************
79   *     Funcion: interpola
80   * Descripcion: Dado el vector con los valores de y originales (yo),
81   *              devolveremos el mismo vector, pero sin errores.
82   *               El algoritmo de interpolacion esta basado en Splines
83   *               Cubicas Naturales.
84   *    Entradas: num_ptos --> Numero de puntos
85   *               yo --> Vector con los valores en el eje Y.
86   *     Salidas: Ninguna.
87   *****************************************************************/
88  void interpola (int num_ptos, double yo[])
89  {
90    int i, j, k, l, m;
91    double ax[MAX_PTOS], bx[MAX_PTOS], cx[MAX_PTOS], dx[MAX_PTOS];
92    double ay[MAX_PTOS], by[MAX_PTOS], cy[MAX_PTOS], dy[MAX_PTOS];
93    double der[MAX_PTOS], gam[MAX_PTOS], ome[MAX_PTOS];
94    double x[MAX_PTOS], y[MAX_PTOS];
95    double t, dt;
96    int ierror, ferror; /* Inicio y fin de cada segmento que tiene error */
97
98    /* Creamos los vectores x, e y sin insertar en ellos los errores */
99    dt = 1./(double) InfoFichero.frames_seg;
100
101   if (arregla_extremos (num_ptos, yo))
102     mostrar_popup ("Demasiados Errores.\n No se puede interpolar
103       automaticamente.",
104                    GTK_MESSAGE_ERROR, GTK_BUTTONS_CLOSE);
105   for (i=0, j=0, t=0; i<num_ptos; i++) {
106     if (yo[i]<999998) {
107       y[j]= yo[i];
108       x[j++]=t;
109     }
110     t+=dt;
111   }
112
113   m = j-1;       /* m es el numero de intervalos que tendremos */
114
115   /* Ahora hallamos la spline que pasa por esos puntos */
116   /* Calculamos el valor de gamma (sera el mismo en X y en Y) */
117   gam[0] = .5;
118   for (i=1; i<m; i++) gam[i] = 1./(4.-gam[i-1]);
119   gam[m] = 1./(2.-gam[m-1]);
120
121   /* Calculamos el valor de omega para abcisas */
122   ome[0] = 3.*(x[1]-x[0])*gam[0];
123   for (i=1; i<m; i++) ome[i] = (3.*(x[i+1]-x[i-1])-ome[i-1])*gam[i];
```

```
124     ome[m] = (3.*(x[m]-x[m-1])-ome[m-1])*gam[m];
125
126     /* Valor de la primera derivada en los puntos (eje X) */
127     der[m]=ome[m];
128     for (i=m-1; i>=0; i=i-1) der[i] = ome[i]-gam[i]*der[i+1];
129
130     /* Sustituimos los valores de gamma, omega y la primera derivada
131        para calcular los coeficientes a, b, c y d */
132     for (i=0; i<m; i++) {
133       ax[i] = x[i];
134       bx[i] = der[i];
135       cx[i] = 3.*(x[i+1]-x[i])-2.*der[i]-der[i+1];
136       dx[i] = 2.*(x[i]-x[i+1])+der[i]+der[i+1];
137     }
138
139     /* Calculamos omega para el eje de ordenadas */
140     ome[0] = 3.*(y[1]-y[0])*gam[0];
141     for (i=1; i<m; i++) ome[i] = (3.*(y[i+1]-y[i-1])-ome[i-1])*gam[i];
142     ome[m] = (3.*(y[m]-y[m-1])-ome[m-1])*gam[m];
143
144     /* Hallamos el valor de la primera derivada... */
145     der[m]=ome[m];
146     for (i=m-1; i>=0; i=i-1) der[i] = ome[i]-gam[i]*der[i+1];
147
148     /* Valor de los coeficientes a, b, c y d en el eje Y */
149     for (i=0; i<m; i++){
150       ay[i] = y[i];
151       by[i] = der[i];
152       cy[i] = 3.*(y[i+1]-y[i])-2.*der[i]-der[i+1];
153       dy[i] = 2.*(y[i]-y[i+1])+der[i]+der[i+1];
154     }
155
156     ierror = ferror =  -1;
157     j = 0; /* Este j identifica el segmento del sistema ya interpolado */
158     for (i=0; i<num_ptos; i++) {
159       if (yo[i] > 999998) {
160         if (ierror == -1) ierror = i;
161         ferror = i;
162       }
163       else {
164         if (ierror != -1) {   /* Interpolamos, el intervalo se acabo */
165           l = ierror;
166           for (k=0; k<=ferror-ierror; k++) {
167        /* Es igual a (1/2+ferror+ierror)*(k+1) */
168             t = ((double)(k+1)/(double)(2+ferror-ierror));
169             yo[l++] = (ay[j-1]+by[j-1]*t+cy[j-1]*t*t+dy[j-1]*t*t*t);
170           }
171         ierror = ferror = -1;
172         }
173         j++;
174       }
175     }
176   }
```

Bibliografía

[1] Artoolkit's official site. http://www.hitl.washington.edu/artoolkit/.

[2] Bazar's official site. http://cvlab.epfl.ch/software/bazar/.

[3] Confederación estatal de personas sordas official site. http://www.cnse.es/.

[4] Curso opengl. http://www.frodrig.com/macedoniamagazine/opengl1.htm (buscado en internet) [accedido en enero,2009].

[5] Diccionario online de la real academia española: http://www.rae.es.

[6] Ffmpeg cross-platform solution: http://ffmpeg.org/ official web.

[7] Gandalf's official site. http://gandalf-library.sourceforge.net/.

[8] Neatvision's official site. http://neatvision.eeng.dcu.ie/.

[9] Open source iniciative's open source definition. http://www.opensource.org/docs/osd.

[10] Opencv's official site. http://opencvlibrary.sourceforge.net/.

[11] Wikipedia web page. http://www.wikipedia.com/.

[12] N. Adamo-Villani, E. Carpenter, and L. Arns. An immersive virtual environment for learning sign language mathematics. pages 20–26, 2006.

[13] Simon Baker, Daniel Scarstein, J.P. Lewis, Stefan Roth, Michael J. Black, and Richard Szeliski. A database and evaluation methodology for optical flow. *Proceedings of the IEEE International Conference on Computer Vision*, 2007.

[14] J. Barron, D. Fleet, and S. Beauchemin. Performance of optical flow techniques. *International Journal of Computer Vision*, 12(1)(43-77), 1994.

[15] Michael J. Black and Yaser Yacoob. Recognizing facial expressions in image sequences using local parameterized models of image motion. *Xerox Palo Alto Research Center and Computer Vision Laboratory, University of Maryland.*

[16] Ángel Herrero Blanco. *Escritura Alfabética de la Lengua de Signos Española*. Universidad de Alicante, 2003.

[17] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2.1 (6th Edition)*. Addison-Wesley Professional, 2007.

[18] G. Bradshaw. *Non-contact surface geometry measurement techniques*. Image Synthesis Group, Trinity College, Dublin, Irlanda, 1999.

[19] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV Library*. O'Really, September, 2008.

[20] Francisco Ramírez C., Klaus Rall, F. Eugenio López G., and Rafael Colas O. Maquinado de una sucesión de curvas. *Ingenierías*, (IV):35–36, 2001.

[21] D. Cristinacce and T.F. Cootes. Feature detection and tracking with constrained local models. *Proc. British Machine Vision Conference*, vol. 3(929-938), 2006.

[22] José Julián Díaz Díaz, Donald Hearn, and M. Pauline Baker. *Gráficas por computadora*. 1995.

[23] P. Ekman and W. Friesen. Facial action coding system: A technique for the measurement of facial movement. *Consulting Psychologists Press, Palo Alto*, 1978.

[24] Yikai Fang, Jian Cheng, J. Wang, K. Wang, J. Liu, and H. Lu. Hand posture recognition with co-training. *ICPR 2008, Pattern Recognition 2008*, 2008.

[25] D. M. Gavrila and Wihelm Runge St. The visual analysis of human movement: A survey. *Computer Vision and Image Understanding*, 3(1), January, 1999.

[26] J. Glauert, R. Kennaway, R. Elliott, and Theobald B. Virtual human signing as expressive animation. *In Symposium on Adaptive Agents and Multi-Agent Systems, Leeds, UK.*, pages 88–106.

[27] Vanessa Herrera Tirado. *GANAS: Generador Automático del Lenguaje de Signos*. Escuela Superior de Informática de Ciudad Real (UCLM), 2007.

[28] B. Horn and B. Schunck. Determining optical flow. *Artificial Intelligence*, 17:185–203, 1981.

[29] A. Just, Y. Rodriguez, and S. Marcel. Hand posture classification and recognition using the modified census transform. *Proceedings of AFGR*, (351-356), 2006.

[30] Mark J. Kilgard. *The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3*. Silicon Graphics, Inc., 1996.

[31] M. Kolsch and M. Turk. Robust hand detection. *Proceedings of AFGR*, 614-619, 2004.

[32] Rung-Huei Liang and Ming Ouhyoung. A real-time continuous alphabetic sign language to speech conversion vr system. *Computer Graphics Forum*, 14:67–77, August 1995.

[33] R. Lienhart, A. Kuranov, and V. Pisarevsky. Empyrical analysis of detection cascades of boosted classifieres for rapid object detection. *Micriprocessor Research Lab Technical Report, Intel Labs*, 2002.

[34] S. Liwicki and M. Everingham. Automatic recognition of fingerspelled words in british sign language. *Proceedings of the 2nd IEEE Workshop on CVPR for Human Communicative Behavior Analysis (CVPR4HB'09)*, pages 50–57, June 2009.

[35] Frank D. Luna. *Introduction to 3D Game Programming with DirectX 9*. Wordware Publishing Inc., 2003.

[36] Tomas Moller, Eric Haines, and Tomas Akenine-Moller. *Real Time Rendering (Second Edition)*. AK Peters, Ltd., 2002.

[37] University of Leeds. Opencv guide. http://www.comp.leeds.ac.uk/vision/opencv/ (buscado en internet) [accedido en enero,2009].

[38] Carme Armentano Oller, Antonio M. Corbí Bellot, Mikel L. Forcada, Mireia Ginestí, Marco A. Montava, Sergio Ortiz, J.A. Pérez Ortiz, Gema Ramírez, and Felipe Sánchez. Apertium, una plataforma de código abierto para el desarrollo de sistemas de traducción automática. *Departament de Llenguatges i Sistemes Informàtics, Universitat d'Alacant*, 2004.

[39] P.Buehler, M. Everingham, D.P. Huttenlocher, and A. Zisserman. Long term arm and hand tracking for continuous sign language tv broadcasts. *Proceedings of the 19th British Machine Vision Conference (BMVC2008)*, pages 1105–1114, September 2008.

[40] P.Buehler, M. Everingham, and A. Zisserman. Learning sign language by watching tv. *Proceedings of the IEEE International Conference on Computer Vision and Pattern recognition (CVPR2009)*, pages 2961–2968, June 2009.

[41] Ignacio Mantilla Prada. *Análisis numérico*. 2004.

[42] F. Remondino, N. D'Apuzzo, and G. Schrotter. Markeless motion capture from single or multi-camera video sequence. *IGP- ETH Zurich, CH*, 2004.

[43] R.Varas. Optical flow: http://www.elai.upm.es/spain/investiga/ gcii/personal/rvaras/opticalflow.htm (buscado en internet) [accedido en octubre, 2009].

[44] M. Sonka, V. Hlavac, and R. Boyle. *Image processing, analysis and machine vision*. Chapman Hall, Londres, Reino Unido, 1993.

[45] Rita Street. *Computer Animation:A Whole New World*. Rockport Publisheres Inc., 1998.

[46] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. *IEEE CVPR*, pages 511–518, 2001.

[47] T. Y. Young and K. S. Fu. *Handbook of pattern recognition and image analysis*. Academic Press, Nueva York, 1986.