Departamento de Inteligencia Artificial
Colmenarejo

PROYECTO FIN DE CARRERA

Time-Of-Flight Camera

Autor: Daniel Sánchez del Álamo Benguigui

Tutor: Miguel Ángel Patricio Guisado

Colmenarejo, 10 de Agosto de 2010

Título: Time-Of-Flight Camera
Autor: Daniel Sánchez del Álamo Benguigui
Director:

EL TRIBUNAL

Presidente: _____

Vocal: _____

Secretario: _____

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día __ de _____ de 20__ en Colmenarejo, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de

VOCAL

SECRETARIO PRESIDENTE

Agradecimientos

Agradezco, en primer lugar al Grupo de Inteligencia Artificial Aplicada (GIAA) con todos y cada uno de sus miembros y, más concretamente a Miguel Ángel Patricio Guisado, la oportunidad ofrecida de realizar un proyecto acorde a mis necesidades e inquietuides que han permitido desarrollar un estudio sobre un área especialmente afín a mis conocimientos.

En segundo lugar, me gustaría agradecer los ingentes esfuerzos realizados por mi familia para que haya podido completar mis estudios fuera de mi hogar.

Resumen

El presente proyecto ha implicado la realización de una API clara, interoperable y eficaz para permitir el uso de cámaras TOF. Este tipo de cámaras permiten capturar fotogramas en un modelo tridimensional que puede ser usado como sustitutivo o extensión del cuerpo humano o de sus habilidades.

Para ello se ha realizado un estudio sobre la tecnología que soporta la cámara y sus posibles aplicaciones a entornos de diferentes naturalezas. Entre dichos entornos se encuentra el ocio digital y las aplicaciones orientadas a maximizar los rendimientos de deportistas de élite.

Para testear los estudios realizados en la tecnología y más concretamente en la cámara designada para el proyecto, se ha implementado una aplicación informática que ha servido para observar los comportamientos en los cambios de los parámetros internos de la cámara.

Se ha desarrollado una prueba de concepto que engloba un volante virtual incluido en un videojuego de coches de carrera donde el usuario gría dicho volante virtual. Las guías para la conducción del vehículo han tenido que crearse como método para el manejo del volante ya que el usuario debe poder encontrar toda la funcionalidad de un vehículo con tan sólo sus dos manos.

Palabras clave: TOF Kinect 3D imagen escáner XBOX XNA XAML WPF DirectX volante virtual cámara tiempo de vuelo interfaz gráfica GUI juego videojuego tecnología LED distancia amplitud sensor triangular tridimensional

Abstract

This project involved the creation of clear API, interoperable and effective to allow the use of TOF cameras. Such cameras can capture frames in a three-dimensional model can be used as replacement or extension of the human body or skills.

This has made a study on the technology behind the camera and its potential application to environments of different natures. Among such environments is the digital entertainment-oriented applications and maximize returns on elite athletes.

To test the technology studies and more specifically in the camera designated for the project, we have implemented an application that has been used to observe the behavior of changes in internal parameters of the camera.

We have developed a proof of concept that encompasses a virtual steering wheel included in a race car videogame where the user gary said virtual wheel. Guidelines for driving the vehicle had to be created as a method to the steering wheel as the user should be able to find all the functionality of a vehicle with only two hands.

Keywords: TOF Kinect 3D image scanner XBOX XNA XAML WPF XAML DirectX virtual camera time-of-flight interface graphic UI GUI UX game videogame technology LED distance amplitude sensor triangular three-dimensional steering wheel

Índice general

Índice de figuras

Índice de códigos

Capítulo 1

Introducción y objetivos

1.1 Claves

La claves para este capítulo son:

- Contextualizar el problema que se presenta.
- Definir los objetivos perseguidos.
- Describir los medios empleados.
- Describir la planificación a seguir.
- Describir la estructura de la memoria.

1.2 Introducción

En la actualidad, las necesidades impuestas por la sociedad son cada vez mayores y requieren para ello, de tecnologías y conocimientos más complejos, que den respuestas a dichas peticiones.

Conceptos hasta hace poco de ciencia-ficción son ahora realidades tangibles y computables que conforman empresas y acciones realizadas por máquinas para los humanos. Un ejemplo de ello, y cada vez más en auge, son las redes sociales, que no es más que una forma expansiva de comunicación actualizada entre seres humanos. El hecho de poder consumir fragmentos de información en cualquier momento es el eje que mueve, motiva y articula las redes sociales.

Este proyecto nace de la necesidad que se tiene de poder modificar el entorno o al menos analizarlo, como lo haría un humano en las mismas condiciones. ¿Cómo lograr una modificación del entorno, previo análisis del mismo? Esta cuestión plantea una serie de dificultades o barreras tecnológicas que el conocimiento moderno debe superar. Con modificación del entorno se pretende mostrar el conjunto de tecnologías que se han desarrollado con carácter extensorio de las cualidades humanas. Una cámara que graba es una extensión de la visión. Una grúa que mueve carga en un muelle es la extensión de un brazo. Una cámara térmica es una extensión del sentido del tacto. Con estas tecnologías se captura y analiza el entorno.

Lo importante es ofrecer una respuesta inteligente en base a dicho análisis. Este es el objetivo de la Inteligencia Artificial que se define como la ciencia que se encarga de realizar procesos bajo una infraestructura determinada que almacena alguna forma primitiva o compleja de conocimiento, y que provoca las acciones a desempeñar.

Este estudio se nutre tanto de la Inteligencia Artificial como de una tecnología moderna que intenta suplantar o complementar la visión. Como todas las ciencias, la Inteligencia Artificial se encuentra fragmentada en áreas temáticas o de aplicabilidad conceptual o teórica diferenciadas del resto. Una de estas áreas es la Visión Artificial o Visión por Computador que intenta capturar y analizar el entorno por medio de cámaras de distinta naturaleza para ofrecer respuestas en base a estímulos acaecidos en dicho entorno.

Por tanto, el marco donde se sitúa el estudio contempla la Inteligencia Artificial y más concretamente, la Visión Artificial. En cuanto a la tecnología empleada como elemento extensivo de la cualidad humana, se emplea una cámara que captura la estructura tridimensional del medio gracias a un conjunto de sensores infrarrojos. Con estos dos elementos se pretende analizar la estructura tridimensional de la anatomía humana para posteriormente detectar los movimientos y provocar las modificaciones pertinentes.

1.2.1 Contexto actual

En este primer capítulo del documento se presenta una tecnología innovadora que puede ser el futuro de muchas aplicaciones informáticas. Esta tecnología conocida como TOF (*Time-Of-Flight* o tiempo de vuelo) representa una forma de captar el mundo de forma tridimensional a través de LEDs instalados en una cámara.

Posteriormente se ofrece un ejemplo comercial e internacional de la aplicación de dicha tecnología orientada al mundo del ocio digital por parte de Microsoft para su consola XBOX 360.

A continuación se hace un breve repaso del estado actual de la tecnología, el funcionamiento intrínseco de la misma y el grado de madurez que presenta. Más concretamente se hace hincapié en el modelo adquirido para el desarrollo del proyecto y las características internas y externas que ofrece la cámara. La cámara empleada para desarrollar el proyecto se muestra en la siguiente imagen:

Figura 1: Kit completo de la cámara SR4000 de Mesa Imaging.

Existen en la actualidad muchas empresas que se están dedicando a la construcción de cámaras TOF lo que es un buen indicativo tanto económico como empresarial de que dichos sistemas pueden formar parte en más o menos tiempo de nuestro presente o futuro.

El eje vertebral de esta sección se encamina a mostrar un ejemplo empresarial de la aplicación de cámaras TOF orientadas al ocio digital y cómo esta industria se nutre del conocimiento y de la tecnología para crear un mercado cada más en auge.

El ejemplo se basa en el proyecto llevado a cabo por Microsoft denominado Natal que ha concluido con la presentación del producto Kinect. Microsoft posee una consola de creación propia llamada XBOX 360 al cual le ha diseñado un complemento ideal para videojuegos más interactivos. El mando tradicional ha desaparecido para dejar paso a los movimientos del cuerpo del usuario que interactúa con la consola. Permite la detección de dos jugadores en la escena lo que hace a Kinect multijugador y a un precio muy económico. Las economías de escala han hecho posible que una cámara TOF valga 150 dólares en Estados Unidos.

Esta propuesta ha sido llevada a cabo debido al emergente mercado que suscitó en su momento la consola de Nintendo Wii con su mando Wiimote que posee la habilidad de señalar objetos en la pantalla y de detectar movimientos en el espacio.

En base a esta nueva forma de orientar el ocio digital donde los juegos de la Wii están orientados a toda la población y sin establecerse segmentos de mercado, Nintendo ha creado en Microsoft un hueco tecnológico que espera tapar con la entrada Kinect.

Figura 2: Kinect para la consola XBOX 360.

1.2.2 Cámara TOF SR4000/SR4K

En esta sección se describen las características propias de la cámara que se va a emplear para el desarrollo del proyecto. Para comenzar con la descripción de la cámara se presentan las dos regiones más importantes que son:

- Capa de iluminación.
- Filtro óptico.

La capa de iluminación presenta una matriz compuesta por LEDs que reciben la información del exterior, mientras que el filtro óptico capta las frecuencias de luz de la escena actual.

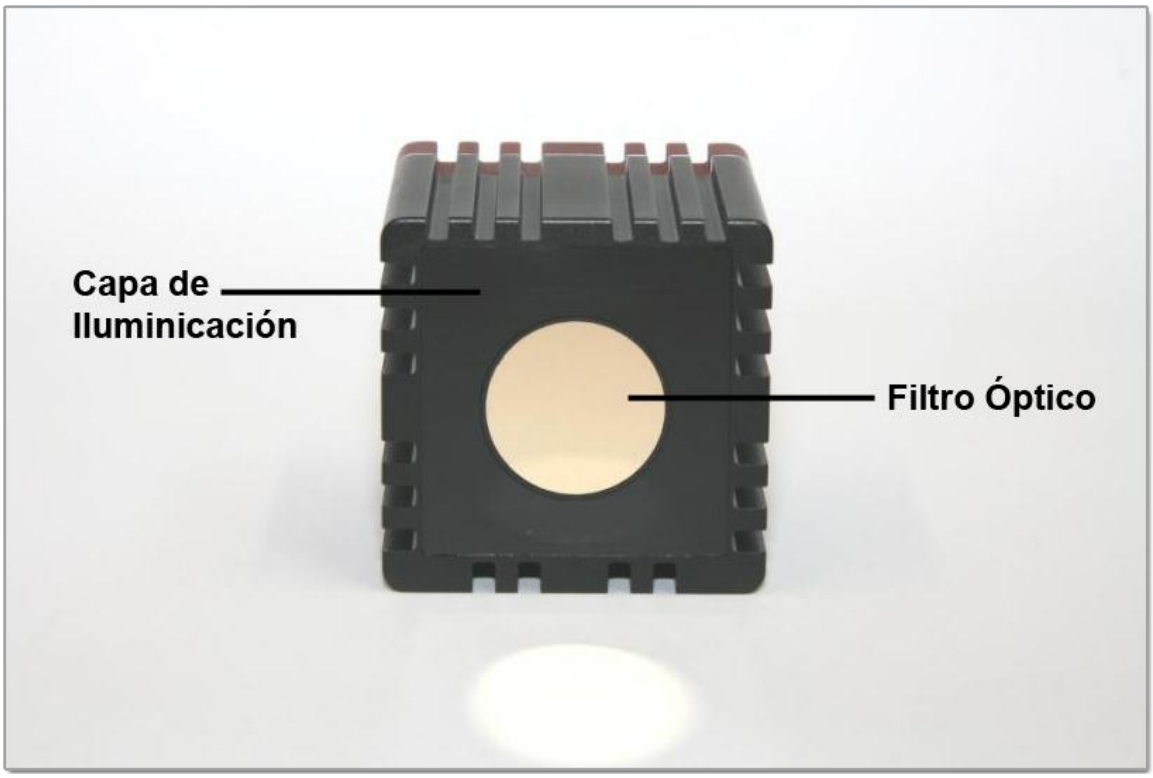


Figura 3: Regiones de captación de la cámara.

A continuación se muestran las conexiones que tiene incorporadas la cámara que determinará la interfaz de comunicaciones desde el ordenador personal a la cámara. Las interfaces de comunicación pueden ser Ethernet o USB. La cámara con la cual se ha desarrollado el proyecto tiene la entrada de comunicaciones Ethernet.

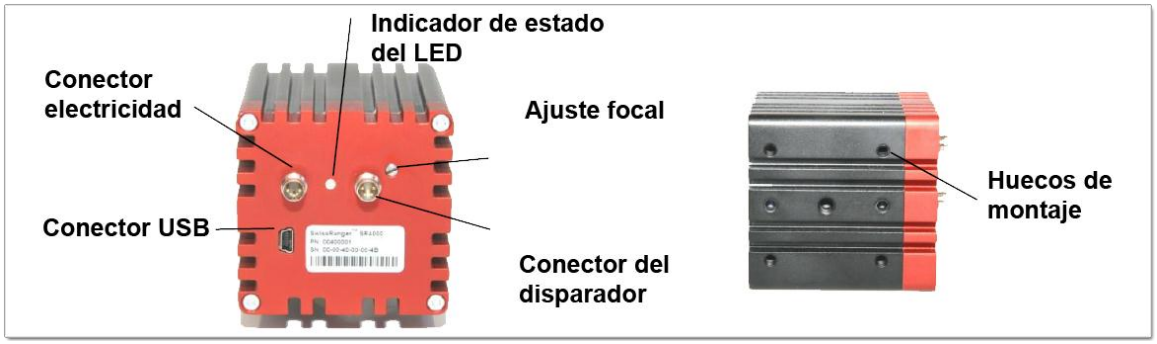


Figura 4: Conectores de la cámara.

En la Figura 4 se muestran los conectores que posee la cámara en versión USB.

El funcionamiento de las cámaras TOF se rige por una fórmula de emisión de un fotón de luz de ida y regreso, una vez rebotado el fotón con el cuerpo u objeto que se quiere detectar.

La fórmula es la siguiente para un único LED:

$$D = \frac{c}{2 \cdot f}$$

Figura 5: Ecuación general de la distancia.

Siendo D la distancia entre la cámara o emisor de luz y el objeto que se quiere detectar, c la velocidad de luz (300.000 km/s) y f la frecuencia de emisión de luz. Con esta fórmula se puede obtener, por tanto, la distancia máxima a la cual pueden estar los objetos para ser procesados por la cámara. Si la frecuencia de emisión de luz es de 30Mhz (frecuencia empleada por defecto en la cámara asignada al proyecto) nos encontramos con la siguiente expresión:

$$D = \frac{c}{2 \cdot f} = \frac{300.000.000\ m/s}{2 \cdot\ 30.000.000\ s^{-1}} = 5\ m$$

Figura 6: Ejemplo de la ecuación general de la distancia.

La información que obtiene por tanto un LED sobre una región asignada es la distancia que existe entre el objeto y el LED. Para poder hallar las tres dimensiones de la región asignada al LED se requiere de al menos dos LEDs por región. Así se podrá triangular con las distancias obtenidas. Si en lugar de aplicar dos LEDs para una región, se aplicase toda la matriz de LEDs, se obtendría una medida aparentemente más precisa.

Para ello la cámara realiza la mediana de todas las triangulaciones de todas las regiones captadas en un tiempo denominado *Tiempo de integración*. A este tiempo hay que sumarle el consumido tras aplicar ciertos filtros a la imagen obtenida. Esto implica que si el tiempo de integración es pequeño, no se aplicarán todas las medianas a todas las regiones y se encontrarán pixeles con incertidumbre.

Otro parámetro importante es la resolución o calidad a la hora de diferenciar entre dos regiones muy juntas en el espacio. Esta resolución viene determinada por la velocidad del procesador de la cámara.

$$D = \frac{c}{2 \cdot f} = \frac{300.000.000\ m/s}{2 \cdot\ 2.800.000.000\ s^{-1}} = 0{,}053\ m$$

Figura 7: Ejemplo de cálculo de la resolución para procesador a 2.8 GHz.

Seguidamente se describe el proceso de captación de imágenes por parte de la cámara usando para ello su API interna.

El proceso de captación de la cámara conlleva tres pasos a lo largo del tiempo que son:

- Adquisición y lectura: la adquisición de la escena se realiza mediante la llamada a la API de la función *SR_Acquire ()* que se explica más adelante. Este paso consiste en obtener las medidas de las distancias de cada uno de los LEDs sin

aplicar cálculo alguno. Este proceso, dependiendo del procesador de la cámara, puede consumir unos 30 nanosegundos basándose en la fórmula de la Figura 5.

- Cálculo: en este paso la cámara realiza las triangulaciones necesarias en base al tiempo de integración y aplica los cálculos de las medianas que pueda en función del número de vecinos asignados progresivamente. Posteriormente se encarga de aplicar los filtros impuestos por el programador a la captación de la imagen.
- Procesar la salida: una vez todos los canales están procesados y los filtros aplicados se activa un *flag* que determina que se ha finalizado con la acción *SR_Acquire ()*. El usuario está ahora en disposición de recoger la información para su lectura y procesamiento posterior.

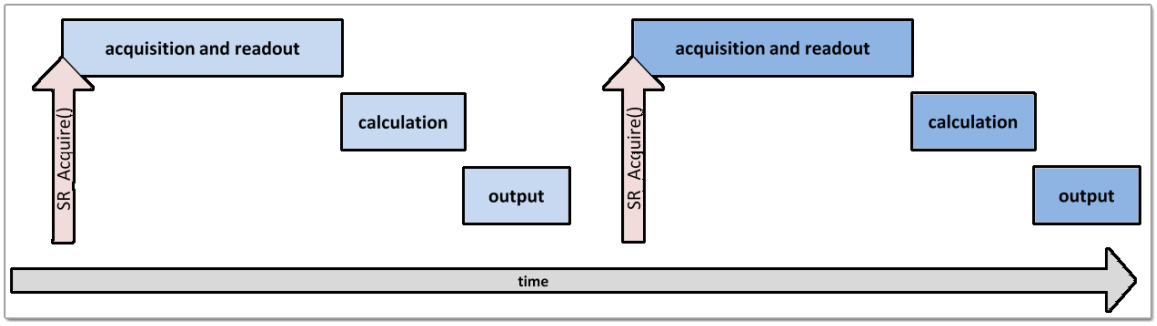


Figura 8: Proceso de captación de la cámara.

1.3 Objetivos

El objetivo fundamental de la tesis es el diseñar una API interoperable y eficaz para interactuar con una cámara TOF. En base a ese objetivo principal, se proponen los siguientes objetivos parciales:

- Investigar la tecnología TOF y sus aplicaciones.
- Investigar la cámara TOF SR4000.
- Diseñar y documentar una API de comunicaciones con la cámara TOF.
- Implementar una aplicación TOF que haga uso de la API.
- Implementar una aplicación orientada al ocio con la cámara SR4000.
- Formalizar la memoria del proyecto.

1.4 Fases del desarrollo

El proyecto ha sido dividido en varias fases de desarrollo partiendo de un modelo en cascada debido principalmente a la dependencia que tienen todas las fases sobre la primera, que supone el conocimiento e investigación de una tecnología que aún está por desarrollar y madurar.

Las fases en las cuales se ha dividido el proyecto han sido:

- Investigación de la tecnología TOF.
- Investigación de la API de la cámara.
- Diseño de una API interoperable.
- Implementación de interfaces de usuario y pruebas de concepto.
- Documentación del proyecto.

1.4.1 Investigación de la tecnología TOF

En esta fase inicial del proyecto se ha tenido que investigar el funcionamiento abstracto de la tecnología TOF, de sus posibles aplicaciones y de la necesidad de seguir apostando por dicha tecnología como pilar estructural de futuras aplicaciones informáticas.

Esta tecnología debe ofrecer suficiente industria y conocimiento como para sustentar posibles inversiones e iniciativas empresariales que garanticen el futuro de la misma.

Para ello se han realizado estudios sobre su génesis, los costes basados en economías de escala y como la industria va supliéndose de dicha tecnología. Un ejemplo explicado anteriormente es por parte de Microsoft en su proyecto Natal que ha desembocado en el producto final Kinect para su consola de ocio digital XBOX 360.

1.4.2 Investigación de la API de la cámara

Una vez decidida que la tecnología TOF puede suponer un cambio drástico en la manera en que las aplicaciones informáticas se desarrollan, comercializan y permiten su interacción con el usuario, se debe investigar la forma o el camino para su industrialización.

Para ello, se ha contado con el modelo de cámara TOF SR4000 de la empresa Mesa Imaging. El objeto, por tanto de esta fase, ha sido conocer más concretamente cómo funciona el modelo de cámara descrita, los parámetros internos de elaboración, ejecución y captura de la escena real, y su posterior análisis para una aplicación basada en tiempo real.

Con una arquitectura de funcionamiento clara y, establecidos los tiempos máximos de respuesta por parte de la cámara, se procede a la siguiente fase del proyecto.

1.4.3 Diseño de una API interoperable

En este punto de ejecución del proyecto se ha obtenido suficiente información tanto matemática, como física o a nivel de código, como para desarrollar una API o librería que maximice, por un lado la funcionalidad ofrecida por la cámara, como que permita el máximo grado de interoperabilidad entre distintas arquitecturas o lenguajes de programación.

Para lograr el objetivo de maximizar ambas premisas, se ha decidido crear una envolvente de las estructuras programadas en C bajo la arquitectura .NET. Con esta librería se ha desarrollado una aplicación que utilice dicha librería y se pueda observar la funcionalidad de la cámara de manera directa.

1.4.4 Implementación de interfaces de usuario y pruebas de concepto

Con una API desarrollada en C# bajo la arquitectura .NET se ha decidido crear una capa abstracta que represente un objeto de la realidad y que pueda ser controlado por el usuario a través de la cámara.

Dicho objeto que representa la realidad ha sido el volante de un vehículo. Para conseguir un efecto de realidad aumentada se ha introducido dicha capa abstracta que represente el volante dentro de un videojuego programado en XNA.

Con esta acción de introducir el volante en el juego se consolidan dos objetivos básicos del proyecto. Por un lado, contribuir al desarrollo de una API interoperable entre diferentes plataformas o arquitecturas, y por otro, crear una capa de abstracción o conocimiento sobre la cámara.

1.4.5 Documentación del proyecto

Con todas las pruebas de concepto realizadas se procede a la documentación de las mismas partiendo de una tecnología innovadora que ofrecerá sus rendimientos en poco tiempo, donde el ejemplo más actual y claro disponible radica en el producto Kinect.

1.5 Medios empleados

Los medios que han hecho posible dicho proyecto están clasificados en cuatro familias o ramos que son:

- Hardware.
- Software.
- Inmuebles.
- Personal.

1.5.1 Hardware

Los medios que se han puesto a disposición del proyecto para ejecutarlo correctamente han sido los siguientes:

- Cámara TOF SR4000.
- Ordenador personal.
- Trípode.

1.5.2 Software

Los medios que se han puesto a disposición del proyecto para ejecutarlo correctamente han sido los siguientes:

- Microsoft Visual Studio 2010 Proffesional.
- Microsoft Windows 7.
- Microsoft Word 2010.
- Microsoft Excel 2010.
- Microsoft Project 2010.
- Microsoft PowerPoint 2010.

1.5.3 Inmuebles

Los medios que se han puesto a disposición del proyecto para ejecutarlo correctamente han sido los siguientes:

- Laboratorio de trabajo en el GIAA (Grupo de Inteligencia Artificial Aplicada).

1.5.4 Personal

Los medios que se han puesto a disposición del proyecto para ejecutarlo correctamente han sido los siguientes:

- Ingeniero en Informática.

1.6 Estructura de la memoria

Esta sección del capítulo se describe cómo será la organización y la estructura de cada uno de los capítulos posteriores a éste. Los capítulos que forman el presente documento son:

- Capítulo 1: Introducción y objetivos.
- Capítulo 2: Análisis de la API de la cámara.
- Capítulo 3: Interfaz gráfica.
- Capítulo 4: Aplicaciones de cámaras TOF al ocio.
- Capítulo 5: Conclusiones.
- Capítulo 6: Presupuesto.

Finalmente se presenta un glosario de términos relevantes y el conjunto de referencias visitadas y leídas que han ayudado a hacer posible este proyecto.

La estructura de cada uno de los capítulos se ha establecido de la misma forma. Todos comienzan con un apartado donde se muestran los objetivos a cumplir o requerimientos que se deben aprender por parte del lector. Esta sección se denomina Claves.

La siguiente sección común a todos los capítulos es la Introducción que sirve para plasmar o extender las claves del capítulo. En esta introducción se desarrollan algunos puntos vistos en capítulos anteriores que sirve para enlazar tales conceptos con el actual capítulo, o bien se aclaran pretensiones o contextos para la finalidad del capítulo.

Las siguientes secciones que presenta cada capítulo serán inherentes al mismo, y prácticamente únicas en cuanto a conceptos o ideas plasmadas, si bien puedan compartir alguna tecnología mostrada.

Para facilitar la lectura de la memoria se incluye a continuación un breve resumen de cada capítulo.

1.6.1 Análisis de la API de la cámara

En este capítulo se presenta una librería que contiene varios elementos de programación, tales como clases, estructuras y enumeraciones, que sirven para interactuar con la cámara TOF desde lenguajes programados bajo la arquitectura .NET.

Esencialmente se explican las diferencias existentes entre la clase de bajo nivel *SRCamAPI* que actúa como envolvente de las llamadas a la cámara en el lenguaje escrito en C, partiendo para ello de los tres archivos de conexión: *libMesaSR.lib, libMesaSR.h* y *definesSR.h*, y la clase orientada a objetos de alto nivel *SRCamera*.

Para entender el comportamiento de estas dos clases se presentan los tipos de datos básicos y enumeraciones empleados por la cámara, las estructuras de almacenamiento de las imágenes en memoria, y las estructuras de control de activación o desactivación de canales y filtros de captación de la escena.

1.6.2 Interfaz gráfica

En este capítulo de la memoria se presenta una aplicación realizada para el manejo de la cámara TOF instalada.

Se pretende formalizar una estructura de trabajo y gestión de los fotogramas captados por la cámara para posteriormente aplicar técnicas de extracción de conocimiento. Para ello se han estudiado los canales y filtros más interesantes para su aplicación y se ha propuesto un modelo de interacción con la cámara.

Otros de los objetivos de la interfaz gráfica es demostrar que la arquitectura .NET puede ser candidata a construir aplicaciones que no exijan rendimientos muy altos, aprovechando su facilidad de programación y obtención de funcionalidad en cortos periodos de tiempo.

Por último se presenta la tecnología para el renderizado de elementos visuales desarrollada por Microsoft que facilita la programación y agiliza los cálculos de renderizado, ya que es ahora *DirectX* y la tarjeta gráfica quienes se encargan de dicha tarea, y no el procesador.

1.6.2.1 Aplicación de cámaras TOF al ocio

En el capítulo 3 se muestra una forma de trabajar con la cámara que es objeto de este actual capítulo, donde ya se organiza una abstracción de la cámara en base a un volante virtual para un juego de carreras de coches programado en XNA.

Se muestran las capturas de pantalla del juego y se describe la utilidad que tiene el volante sobre el videojuego. El valor diferencial introducido por el volante al juego lo hace más real por medio de gradientes de acciones y no acciones activadas o desactivadas. Los gradientes en las acciones implican una intensidad designada a la acción como pueda ser el frenado o la aceleración.

Para finalizar el capítulo se describe la clase Volante en profundidad y se exponen algunos ejemplos donde se ha incluido dicha clase en el código del videojuego.

1.6.3 Conclusiones

En este capítulo se plasman las conclusiones alcanzadas tras analizar los objetivos realizados, y se proponen una serie de avances que se pueden realizar partiendo de lo conseguido en el actual proyecto.

En cuanto a los objetivos, se ha de comentar que se han logrados todos excepto la creación de modelos tridimensionales de usuarios en la escena, ya que la propuesta aunque apreciablemente sencilla conlleva un aumento tanto de recursos materiales como de tiempo asignado al desempeño del proyecto.

Finalmente se exponen las líneas futuras de aplicación de la tecnología estudiada y aplicada en entornos médicos donde la colaboración hombra-máquina es escasa, y donde se puede ayudar a disminuir el riesgo en las operaciones quirúrjicas en mejoras didácticas y tutoradas.

1.6.4 Presupuesto

En este último capítulo de la memoria se describen todos los gastos incurridos en la elaboración del proyecto, y su argumentarción del por qué son necesarios tales gastos en los recursos empleados.

Capítulo 2

Análisis de la API de la cámara

2.1 Claves

La claves para este capítulo son:

- Describir la estructura de un ensamblado.
- Describir el ensamblado *SwissRanger*.
- Diferenciar entre *SRCamera* y *SRCamAPI*.

2.2 Introducción

En este capítulo se describen los conceptos de *Namespace* y *Ensamblado*, y con mayor profundidad el *Namespace SwissRanger* que interactúa con la cámara desde la plataforma .NET. Un namespace o espacio de nombrado único es un contenedor lógico donde se almacenan objetos de diversa naturaleza para mantener un orden estructural acorde a las necesidades de cada aplicación o entorno, donde cada objeto declarado posee un nombre único.

Un único namespace puede contener definiciones de clases, enumeraciones, otros namespaces, estructuras de datos, etc.

Una vez programada la estructura del namespace con todos sus elementos, dicho namespace debe ser compilado en uno o varios archivos denominados ensamblados. Los ensamblados (*Assembly*) son los bloques estructurales fundamentales de la arquitectura .NET conteniendo además de la información compilada del código fuente, meta-información de control de versiones o permisos de seguridad, entre otras utilidades o fuentes de acceso.

El ensamblado que contiene el namespace SwissRanger que controla y da acceso a la funcionalidad de la cámara está almacenado en un único fichero *DLL* (*Dynamic Link Library – Librería de Enlace Dinámico*) llamada *SwissRanger.dll*.

Para implementar completamente el proyecto se han requerido de dos ensamblados externos a la API ofrecida por .NET. Uno de ellos es *SwissRanger* que se encarga de gestionar de la funcionalidad de la cámara y otro es *Fluent* que se encarga de mostrar en la interfaz gráfica la barra de menú con el aspecto del *Office 2010*. Más adelante se explicará el porqué de su uso en dicha aplicación.

Antes de pasar a describir el namespace es conveniente tener presente los símbolos manejados por la arquitectura .NET en *Visual Studio* que representan los elementos estructurales del ensamblado. A continuación se detallan los iconos que representan dichos elementos estructurales:

-  Representa un ensamblado o referencia al mismo.
-  Representa una clase.
-  Representa un miembro de una enumeración.
-  Representa una enumeración.
-  Representa un atributo de una clase o estructura de datos.
-  Representa un método público.
-  Representa un namespace.
-  Representa una propiedad. La propiedad en la arquitectura .NET encapsula un *getter* y un *setter*, o un método de asignación al atributo interno encapsulado por la propiedad y un método de obtención del atributo interno, respectivamente.
-  Representa cualquier elemento de forma estática por medio de la palabra reservada *Static*.
-  Representa una estructura de datos. A diferencia de las clases, las estructuras en la infraestructura .NET sólo deberían poseer atributos y propiedades, aunque

puede darse el caso de encontrar estructuras con métodos. Cuando una estructura posee métodos, debería crearse para ello una clase.

-  Representa que el objeto implementado es compatible con el .NET Framework de XNA para desarrollos de juegos en PC o XBOX.

2.3 Ensamblado *SwissRanger*

Como se ha mencionado anteriormente, un namespace puede estar compuesto de uno o varios objetos, clases, namespaces, enumeraciones, estructuras o cualquier otro elemento aceptado por la arquitectura .NET.

En este apartado se describe en profundidad el ensamblado *SwissRanger* que es objeto y motivo fundamental de la elaboración de este proyecto.

La estructura del ensamblado se puede observar en la imagen siguiente donde se muestran todos los elementos organizados y jerárquicos que lo componen:

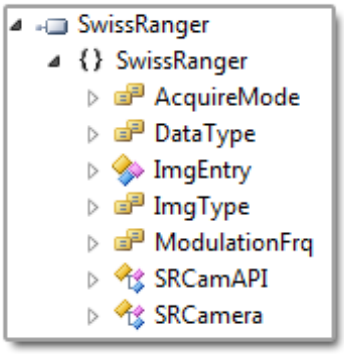


Figura 9: Estructura del ensamblando *SwissRanger*.

2.3.1 Ensamblado *SwissRanger*

Determina el ensamblado completo o el fichero donde se almacena la estructura de datos o código fuente organizado. Como se puede observar, sólo existe un namespace dentro del ensamblado.

Dicho ensamblado se encuentra almacenado en el fichero *SwissRanfer.dll*, lo cual permite que cualquier aplicación pueda hacer uso del mismo.

2.3.2 Namespace *SwissRanger*

Representa el namespace único que gestiona o maneja la cámara. Este namespace contiene las siguientes definiciones de objetos:

- Enumeración *AcquireMode*: Establece el modo de procesamiento de los pixeles para generar un mapa que posteriormente podrá ser mostrado por un control determinado.

- Enumeración *DataType*: Establece un *wrapper* o envolvente para los tipos de datos básicos implementados en sistemas *WINNT/UNIX*.
- Estructura de datos *ImgEntry*: Representa una imagen que contiene un puntero a los pixeles, además de meta-información relacionada con la imagen.
- Enumeración *ImgType*: Determina el tipo de imagen que se ha solicitado a la cámara. El valor de la enumeración actúa como un filtro pre-procesado que la cámara ofrece a los usuarios cuando se solicita un fotograma.
- Enumeración *ModulationFrq*: Establece la velocidad o frecuencia por la cual la cámara capta cada uno de los fotogramas. Estos valores de frecuencia son cruciales cuando se quiere crear una aplicación en tiempo real.
- Clase *SRCamAPI*: Es un *wrapper* estático bajo la arquitectura .NET que encapsula las librerías creadas originalmente en C para el manejo de la cámara. Los ficheros encapsulados son *libMesaSR.lib*, *libMesaSR.h* y *definesSR.h*.
- Clase *SRCamera*: Es una clase que encapsula las llamadas a la clase estática *SRCamAPI* y que ofrece una funcionalidad ordenada a partir de llamadas no estructuradas. Esto permite aplicar todos los conceptos y ventajas de la programación orientada a objetos bajo el paraguas de la infraestructura .NET.

2.3.3  Enumeración *AcquireMode*

La enumeración *AcquireMode* representa los modos de captura que la cámara ofrece al exterior. Se ha de comentar que muchos de los modos de la enumeración son experimentales y otros pertenecen a versiones anteriores de la cámara. Esto reduce el abanico de posibilidades o modos de captura pasando de tener doce filtros a menos de la mitad.

Los modos por defecto que la cámara SR4000 tiene implementados y que se recomiendan cuando se realiza una captura de una imagen son:

AM_COR_FIX_PTRN | *AM_CONV_GRAY* | *AM_DENOISE_ANF*

La colección de filtros que ofrece la cámara es la siguiente:

-  AM_COR_FIX_PRTN: aplica un modo o patrón de corrección del ruido producido por los fotogramas captados. Se recomienda su uso cuando las distancias superen el metro.
-  AM_MEDIAN: aplica la mediana de 3x3 pixeles en torno al píxel calculado.
-  AM_TOGGLE_FRQ: aplica una conmutación en frecuencia desde los 19Mhz hasta los 21Mhz para cada uno de los fotogramas. Sólo se puede usar en el modelo de cámaras TOF SR3000.
-  AM_CONV_GRAY: convierte la imagen en amplitud obtenida multiplicándola por la raíz cuadrada de la distancia, haciéndola más parecida a cámaras de uso cotidiano.

-  AM_SW_ANF: aplica un filtro software de 7x7 pixeles en torno al píxel calculado. Este filtro sólo es para uso experimental y no se recomienda su uso para aplicaciones en producción.
-  AM_RESERVED0: Sólo se puede usar en el modelo de cámaras TOF SR3000, es para uso experimental y no se recomienda su uso para aplicaciones en producción.
-  AM_RESERVED1: se aplica cuando las distancias al objeto son menores de un metro obteniéndose mejores resultados que con el modo AM_COR_FIX_PTRN. Para obtener estos resultados mejorados se debe llamar a la función estática del namespace *SwissRanger.SRCamAPI.SR_CoordTrfFlt ()*.
-  AM_CONF_MAP: se aplica el filtro para obtener el canal que representa la confianza o medida de incertidumbre que presentan el conjunto de pixeles proporcionados por la cámara.
-  AM_HW_TRIGGER: se aplica cuando es la cámara quien controla el proceso de adquisición de los fotogramas y el usuario espera que la cámara le proporcione los fotogramas capturados.
-  AM_SW_TRIGGER: se aplica cuando es el usuario quien controla el proceso de adquisición de los fotogramas por medio de la llamada estática *SwissRanger.SRCamAPI.SR_Acquire ()* o por medio de la llamada al objeto *SwissRanger.SRCamera.Acquire ()*.
-  AM_DENOISE_ANF: aplica un filtro de 5x5 pixeles en torno al píxel calculado.
-  AM_MEDIANCROSS: aplica un filtro de mediana cruzada de 3x3 pixeles en torno al píxel calculado. Este filtro sólo es para uso experimental y no se recomienda su uso para aplicaciones en producción.

2.3.4  Enumeración *DataType*

La enumeración *DataType* encapsula los tipos de datos básicos de un sistema WINNT/UNIX para poder abstraerlos y que la cámara los trate de la misma forma sin tener en cuenta el sistema soportado. Por otro lado, esta enumeración no se utiliza en ninguna llamada dentro del código fuente del proyecto, pero forma parte de la estructura de datos *ImgEntry*.

La colección de tipos de datos básicos que ofrece la cámara es la siguiente:

-  DT_CHAR: encapsula el tipo de datos básico *char*.
-  DT_DOUBLE: encapsula el tipo de datos básico *double*.

-  DT_FLOAT: encapsula el tipo de datos básico *float*.
-  DT_INT: encapsula el tipo de datos básico *int*.
-  DT_NONE: encapsula el tipo de datos básico *void*.
-  DT_SHORT: encapsula el tipo de datos básico *short*.
-  DT_UCHAR: encapsula el tipo de datos básico *unsigned char*.
-  DT_UINT: encapsula el tipo de datos básico *unsigned int*.
-  DT_USHORT: encapsula el tipo de datos básico *unsigned short*. Este atributo se usa por defecto en la estructura de datos *ImgEntry* en el campo o atributo *datatype*.

2.3.5  Estructura de datos *ImgEntry*

La estructura de datos *ImgEntry* se encarga de almacenar los datos capturados por la cámara así como meta-información necesaria en cuanto a las características propias de la cámara.

La estructura está compuesta de los siguientes atributos:

-  data: atributo que viene determinado por un puntero a *void* (*void**) donde se almacenan todos los pixeles capturados por la cámara.
-  imgType: atributo que viene determinado por un valor de la enumeración *ImgType* que se explica más adelante.
-  dataType: atributo que viene determinado por un valor de la enumeración *DataType* anteriormente explicada. El valor usado de este atributo por la librería *SwissRanger.dll* para almacenar todas las estructuras implementadas en el código es DT_USHORT.
-  width: atributo que determina el número de pixeles que tiene la cámara en su coordenada X.
-  height: atributo que determina el número de pixeles que tiene la cámara en su coordenada Y.

2.3.6  Enumeración *ImgType*

La enumeración *ImgType* encapsula los tipos de imágenes que la cámara ofrece al usuario de la misma. Es importante diferenciar correctamente esta enumeración con la

enumeración *AcquireMode* que determina el conjunto de filtros que se aplican al tipo de imagen almacenada en el canal de la distancia.

De toda la colección de canales que la cámara implementa sólo se han hecho uso de los canales IT_DISTANCE, IT_AMPLITUDE e IT_CONF_MAP.

La colección de canales que ofrece la cámara es la siguiente:

-  IT_DISTANCE: este canal es el más importante debido a que la cámara aplica a dicho canal todos los filtros anteriormente explicados en la enumeración *AcquireMode*. Almacena los pixeles en función de la distancia que se encuentran los objetos con respecto de la cámara. Los valores máximos permitidos para cada uno de los pixeles es *0xFFFF* y depende de la frecuencia de captación de la cámara.
-  IT_AMPLITUDE: este canal almacena la información relativa a la amplitud de la cámara, y se comporta como la reproducción de una cámara normal.
-  IT_INTENSITY: este canal almacena la información relativa a la intensidad de la cámara, aunque sólo es válido para el modelo SR2000. Recordar que la cámara utilizada en el proyecto es el modelo SR4000/SR4K.
-  IT_TAP0: este canal se usa para la realización de pruebas y no debería usarse en aplicaciones en producción.
-  IT_TAP1: este canal se usa para la realización de pruebas y no debería usarse en aplicaciones en producción.
-  IT_TAP2: este canal se usa para la realización de pruebas y no debería usarse en aplicaciones en producción.
-  IT_TAP3: este canal se usa para la realización de pruebas y no debería usarse en aplicaciones en producción.
-  IT_SUM_DIFF: este canal realiza una suma diferencia entre los canales almacenados en IT_TAPX de la siguiente forma: (IT_TAP0 + IT_TAP2) – (IT_TAP1 + IT_TAP3).
-  IT_CONF_MAP: este canal almacena información de cada pixel relativa a la incertidumbre que presenta con respecto al conjunto de fotogramas capturados. El valor más alto que se almacena es *0xFFFF*. El grado de incertidumbre que contiene cada pixel ayuda a crear imágenes tridimensionales más eficientes ahorrando cómputos innecesarios si los pixeles presentasen elevadas tasas de error.
-  IT_UNDEFINED: este canal almacena información sobre cualquier tipo de imagen que se quiera, quedando de libre disposición para el usuario o programador.
-  IT_LAST: este canal almacena el tipo de canal usado por última vez por el usuario o programador de la cámara.

2.3.7  Enumeración *ModulationFrq*

La enumeración *ModulatioFrq* encapsula las diferentes frecuencias a las cuales puede capturar la cámara. Teniendo en cuenta el tipo de LED (*Light-Emitting Diode* o Diodo de Emisión de Luz) y su velocidad de propagación, y de la frecuencia a las cuales se irradian las emisiones de luz, se obtiene una medida certera de la distancia a la cual se encuentran los objetos.

El principal problema de esta enumeración se debe a que sólo se pueden usar tres tipos de frecuencia óptima para la cámara SR4000 quedando las demás sin uso o incluso prohibidas por el propio fabricante.

La colección de frecuencias que ofrece la cámara es la siguiente:

-  MF_40MHZ: Esta frecuencia solamente se puede usar para cámaras SR3000. El rango máximo de alcance es de 3.75 metros.
-  MF_30MHZ: Esta frecuencia solamente se puede usar para cámaras SR3000 y SR4000. El rango máximo de alcance es de 5.00 metros.
-  MF_21MHZ: Esta frecuencia solamente se puede usar para cámaras SR3000. El rango máximo de alcance es de 7.14 metros.
-  MF_20MHZ: Esta frecuencia solamente se puede usar para cámaras SR3000. El rango máximo de alcance es de 7.50 metros.
-  MF_19MHZ: Esta frecuencia solamente se puede usar para cámaras SR3000. El rango máximo de alcance es de 7.89 metros.
-  MF_60MHZ: Esta frecuencia solamente se puede usar para cámaras SR4000, pero es de uso interno y no debe ser usada para aplicaciones en producción. El rango máximo de alcance es de 2.50 metros.
-  MF_15MHZ: Esta frecuencia solamente se puede usar para cámaras SR4000, pero es de uso interno y no debe ser usada para aplicaciones en producción. El rango máximo de alcance es de 10.00 metros.
-  MF_10MHZ: Esta frecuencia solamente se puede usar para cámaras SR4000, pero es de uso interno y no debe ser usada para aplicaciones en producción. El rango máximo de alcance es de 15.00 metros
-  MF_29MHZ: Esta frecuencia solamente se puede usar para cámaras SR4000. El rango máximo de alcance es de 5.17 metros.
-  MF_31MHZ: Esta frecuencia solamente se puede usar para cámaras SR4000. El rango máximo de alcance es de 4.84 metros.

-  MF_14_5MHZ: Esta frecuencia solamente se puede usar para cámaras SR4000, pero es de uso interno y no debe ser usada para aplicaciones en producción. El rango máximo de alcance es de 10.34 metros.
-  MF_15_5MHZ: Esta frecuencia solamente se puede usar para cámaras SR4000, pero es de uso interno y no debe ser usada para aplicaciones en producción. El rango máximo de alcance es de 9.68 metros.

2.3.8  Envolvente *SRCamAPI*

Esta clase que forma parte del namespace no contiene llamadas a métodos que se consideren orientadas a objetos, sino más bien, representa una envolvente a las funciones programadas y empaquetadas en los ficheros *libMesaSR.lib*, *libMesaSR.h* y *definesSR.h*.

Este mecanismo de envolver llamadas a funciones programadas en lenguajes no manejados por medio de llamadas estáticas en lenguajes manejados se denomina *P/Invoke* (*Platform Invocation Services*). La peculiaridad que presenta esta envolvente radica en el uso del fichero *libMesaSR.lib*, ya que en cierto modo se esperaba obtener un fichero DLL y no una librería estática.

Las llamadas estáticas presentadas a continuación no especifican completamente la signatura de la función, ya que se encuentra perfectamente definida en la propia librería *SwissRanger.dll*.

Todas las llamadas reciben como parámetro el puntero al manejador de la cámara, ya que éstas llamadas pueden usarse indistintamente para una o varias cámaras a la vez, otorgando así un mayor grado versatilidad por parte de la envolvente.

La colección de métodos estáticos que posee la envolvente *SRCamAPI* es la siguiente:

-  SR_Acquire (): esta llamada provoca que la cámara capture el fotograma de ese momento y transfiera los datos desde la cámara al ordenador. Esta función devuelve el número de bytes transferidos al ordenador o un valor negativo si hubo algún error.
-  SR_Close (): esta llamada cierra el dispositivo abierto por medio del manejador. Los valores que puede devolver la función son:
    - 0: Se cerró correctamente.
    - -1: El dispositivo o manejador no es correcto.
    - -2: No se puede liberar el dispositivo.
    - -3: No se puede cerrar el manejador.
-  SR_CoordTrfDbl (): esta llamada transforma las coordenadas esféricas obtenidas de capturar la imagen en coordenadas cartesianas. Esta función recibe

como parámetros tres *arrays* de *double* donde se almacenen las coordenadas cartesianas y los valores de *pitchx*, *pitchyk* y *pitchz*.

-  SR_CoordTrfFlt (): esta llamada transforma las coordenadas esféricas obtenidas de capturar la imagen en coordenadas cartesianas. Esta función recibe como parámetros tres *arrays* de *float* donde se almacenen las coordenadas cartesianas y los valores de *pitchx*, *pitchyk* y *pitchz*. Aunque el tipo de datos *double* tiene más precisión que el tipo de datos *float*, tanto la función *SR_CoordTrfFlt ()* como *SR_CoordTrfDbl ()* devuelven las mismas coordenadas cartesianas.
-  SR_CoordTrfUint16 (): esta llamada transforma las coordenadas esféricas obtenidas de capturar la imagen en coordenadas cartesianas. Esta función recibe como parámetros tres *arrays* de *short* donde se almacenen las coordenadas cartesianas y los valores de *pitchx*, *pitchyk* y *pitchz*.
-  SR_GetAmplitudeThreshold (): esta llamada obtiene el valor del umbral para el canal en amplitud.
-  SR_GetCols (): esta llamada obtiene el valor de las columnas o el ancho de los fotogramas que la cámara ofrece al usuario.
-  SR_GetDistanceOffset (): esta llamada solamente debe usarse para cámaras SR2000.
-  SR_GetImage (): esta llamada devuelve un puntero a la estructura de datos donde está almacenada la imagen. Para ello requiere de un valor índice que determina el canal de la imagen que devuelve en el puntero.
-  SR_GetImageList (): esta llamada devuelve un puntero a la estructura de datos donde están almacenadas todas las imágenes o canales procesados por la cámara. Para ello requiere de un puntero a una lista de estructuras *ImgEntry*.
-  SR_GetIntegrationTime (): esta llamada obtiene el valor del tiempo de integración que la cámara invierta en procesar los fotogramas capturados. El tiempo de integración depende de cada modelo de cámara y para la SR4000 viene determinado por la siguiente fórmula:

$$0.300\ ms + (intTime) * 0.100\ ms$$

A mayor tiempo de integración (*intTime*) mayor tiempo consume la cámara en procesar la imagen y menor grado de incertidumbre poseen los pixeles. Para un tiempo de integración de 3 ms, el tiempo total de procesado es de:

$$0.300\ ms + 3 * 0.100\ ms = 0.600\ ms$$

-  SR_GetMode (): esta llamada obtiene el conjunto de filtros aplicados por la cámara al canal de distancias en base a la enumeración *AcquireMode*.

-  SR_GetModulationFrequency (): esta llamada obtiene la frecuencia de captación de fotogramas por parte de la cámara en base a la enumeración *ModulationFrq*.
-  SR_GetReg (): esta llamada devuelve el valor del registro pasado por parámetro de la cámara. Los registros de la cámara no han sido utilizados en ninguna parte del proyecto.
-  SR_GetRows (): esta llamada devuelve el valor de las filas o el alto de los fotogramas que la cámara ofrece al usuario.
-  SR_GetVersion (): esta llamada obtiene la versión de la librería *libusbsr.dll*.
-  SR_OpenAll (): esta llamada abre todos los dispositivos conectados al ordenador al mismo. Esta función no debería usarse en aplicaciones en producción.
-  SR_OpenDlg (): esta llamada abre un cuadro de diálogo donde se muestran las cámaras visibles por los distintos medios (Ethernet, USB, etc.) para poder conectarse a ellas. Esta función recibe como parámetro el manejador de la ventana actual de la interfaz gráfica así como un valor que indica el modo de apertura del cuadro de diálogo.
-  SR_OpenETH (): esta llamada conecta directamente con la cámara utilizando para ello una conexión Ethernet. Esta función recibe por parámetro la dirección IP de la cámara para conectarse a ella.
-  SR_OpenSettingsDlg (): esta llamada abre un cuadro de diálogo para modificar los dos argumentos más importantes que posee la cámara: el umbral de la amplitud y el tiempo de integración.
-  SR_OpenUSB (): esta llamada conecta directamente con la cámara a partir de un identificador o número de serie que posee la entrada USB de la cámara.
-  SR_ReadSerial (): esta llamada devuelve el número de serie o ID para las cámaras con conexión USB.
-  SR_SetAmplitudeThreshold (): esta llamada asigna el umbral de amplitud a un valor que viene definido por el parámetro pasado a la función. El valor por defecto para este atributo es cero.
-  SR_SetAutoExposure (): esta llamada activa o desactiva la auto-exposición de la cámara en función de una serie de parámetros pasados a la función. Los parámetros necesarios son:

    - minIntTime: mínimo del tiempo de integración. Si el valor del parámetro *minIntTime* es *0xFF* se desactiva la auto-exposición.
    - maxIntTime: máximo del tiempo de integración.

- percentOverPos: porcentaje de sobre-exposición.
- desiredPos: posición deseada entre 0 y 255.

En la imagen siguiente se muestra la utilidad que tienen los dos parámetros últimos: *percentOverPos* y *desiredPos*.

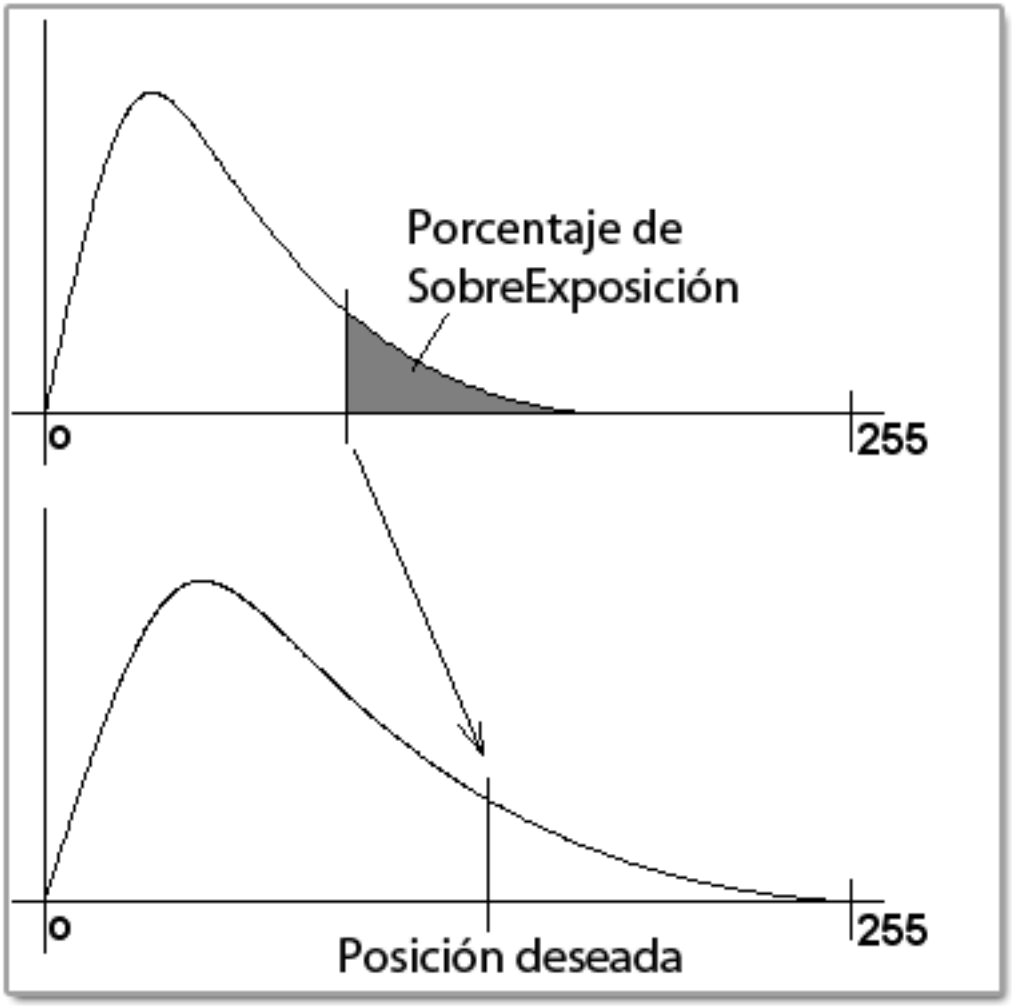


Figura 10: Auto-exposición y los parámetros *percentOverPos* y *desiredPos*.

- SR_SetDistanceOffset (): esta llamada solamente debe usarse para cámaras SR2000.
- SR_SetIntegrationTime (): esta llamada asigna el valor del parámetro pasado al atributo encapsulado que representa el tiempo de integración.
- SR_SetMode (): esta llamada asigna el valor del parámetro pasado al atributo encapsulado que representa el modo basado en la enumeración *AcquireMode*.

- SR_SetModulationFrequency (): esta llamada asigna el valor del parámetro pasado al atributo encapsulado que representa el la frecuencia de captación basado en la enumeración *ModulationFrq*.
- SR_SetReg (): esta llamada asigna el valor del parámetro pasado al atributo encapsulado que representa un registro interno de la cámara.
- SR_SetTimeout (): esta llamada establece el tiempo de retardo en milisegundos que se tarda en poder comunicarse con la interfaz de comunicaciones (USB, Ethernet, etc.) para obtener los datos captados por la cámara.
- SR_StreamToFile (): esta llamada abre un fichero pasado por parámetro para almacenar los datos procedentes de la captación de la cámara. Los modos de apertura son los siguientes:

    - *Open-Create* (0): Abre un fichero y lo crea.
    - *Open-Append* (1): Abre un fichero para añadir datos al final del mismo.
    - *Close* (2): Cierra el fichero.

2.3.9 Clase *SRCamera*

En este apartado ya se describe la clase que hace uso interno de la envolvente *SRCamAPI*. Al tener una clase, se pueden aplicar todos los conceptos clásicos de la programación orientada a objetos tales como el polimorfismo o la herencia para crear estructuras más abstractas y complejas.

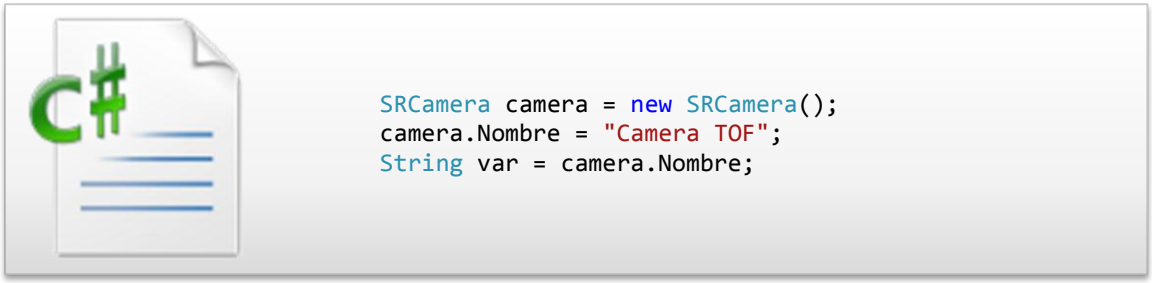
Se comentan las características propias de la clase como los métodos y las propiedades, los usos que se hacen de ella en otras clases derivadas y ejemplos de código para su manejo. Cada apartado tiene una descripción de la propiedad o método y su ejemplo anexo de código.

Antes de pasar a describir cada uno de los elementos que forman parte de la clase *SRCamera*, es conveniente aclarar el concepto de *propiedad*, que sustituye dos métodos básicos en cualquier clase: *get* y *set*. Un ejemplo de ello se presenta a continuación:

```
private String _nombre = String.Empty;
public String Nombre
{
        get { return _nombre; }
        set { _nombre = value; }
}
```

Código 1: Definición de una propiedad.

La primera línea determina el atributo privado *_nombre* encapsulado de la clase, mientras que lo siguiente es la definición de la propiedad *Nombre* con las dos definiciones para la obtención del valor del atributo y para la asignación del mismo. Se presenta a continuación un ejemplo de uso de cada una de las definiciones:



```
SRCamera camera = new SRCamera();
camera.Nombre = "Camera TOF";
String var = camera.Nombre;
```

Código 2: Uso de una propiedad.

Lo importante de la propiedad es que el método aplicable (*get* o *set*) dependerá del lugar que ocupe en el lado de la igualdad. En esta clase todas las propiedades son exclusivamente de lectura, por lo que no presentan el método interno *Set*.

La colección de propiedades que posee la clase *SRCamera* es la siguiente:

- CamPtr: puntero manejador que requiere cada llamada de la envolvente *SRCamAPI* y que identifica a la cámara.
- Height: número de pixeles que tiene la cámara en su componente vertical.
- IsConnected: indica si la cámara está conectada a la interfaz para iniciar la transferencia de datos.
- Serial: número de serie o ID que posee la cámara si la interfaz de comunicaciones es por medio de un dispositivo USB.
- Width: número de pixeles que tiene la cámara en su componente horizontal.

La colección de métodos que posee la clase *SRCamera* es la siguiente:

- Acquire (): este método realiza la captación del fotograma por parte de la cámara y almacena dicha información en los distintos canales aplicando al canal de las distancias los filtros determinados.

  Si el método devuelve un valor negativo, indica que no se ha podido capturar el fotograma actual. Cuando se está leyendo de un fichero, se debe ejecutar el método *Acquire* para simular la captación del fotograma, y al finalizar la lectura del mismo se produce un bloque en la cámara que debe ser controlado.

```
SRCamera camera = new SRCamera();
if (camera.IsConnected)
{
	int result = camera.Acquire();
	if (result < 0)
	{
		MessageBox.Show("Se ha producido un error en
			Acquire.");
	}
}
```

Código 3: Ejemplo de *SRCamera.Acquire ()*.

- Connect (): este método realiza una conexión con la cámara desde cualquier dispositivo o interfaz disponible para ello.

  Si el método devuelve un valor negativo, indica que no se ha podido conectar con la cámara. Este método no se ha usado en la aplicación realizada para el proyecto.

```
SRCamera camera = new SRCamera();
if (camera.Connect() < 0)
{
	MessageBox.Show("Error al conectar con la cámara.");
}
```

Código 4: Ejemplo de *SRCamer.Connect ()*.

- ConnectAll (): este método realiza una conexión propia por cada cámara conectada a una interfaz o dispositivo de comunicaciones. Este método requiere de una dirección IP y una máscara de subred para establecer y buscar todas las cámaras a las cuales se pueda conectar.

  El método devuelve una *array* de cámaras conectadas y listas para comenzar a capturar fotogramas. Este método no ha sido usado en el proyecto ya que sólo existe una cámara como recurso.

```
SRCamera camera = new SRCamera();
List<SRCamera> camaras;
camaras = new List<SRCamera>(camera.ConnectAll(address,
mask));
MessageBox.Show("Se han conectado " + camaras.Count + "
	cámaras.");
```

Código 5: Ejemplo de *SRCamer.ConnectAll ()*.

-  ConnectDialog (): este método abre un cuadro de diálogo para poder conectar de forma gráfica a las cámaras que estén comunicadas con las interfaces de transferencia de datos. Este método está sobrecargado permitiendo un modo de apertura para el cuadro de diálogo que no se recomiendan. Los modos son los siguientes:

    - 0 para abrir el mismo dispositivo que la última vez abierto.
    - 1 para abrir el cuadro de diálogo por defecto.
    - 2 para conectar con la cámara sin tener en cuenta ninguna configuración previa. No se recomienda su uso.

  Como se puede observar en la Figura 3, existen dos botones que permiten escanear (*Scan*) la red en busca de cámaras o bien conectar (*Connect*) con la cámara seleccionada. Se ofrece la posibilidad, si las características de la cámara así lo permiten, poder conectarse mediante una interfaz USB.

  Especial importante adquiere el dispositivo siempre presente denominado *Camera File Stream* que sirve para abrir ficheros guardados con anterioridad que almacenan fotogramas captados por la cámara. De esta forma se crea una cámara virtual que mostrará el mismo flujo de datos que están contenidos en el fichero seleccionado.

  Para poder mostrar los cuadros de diálogo, el método requiere de un puntero a una ventana de la aplicación.

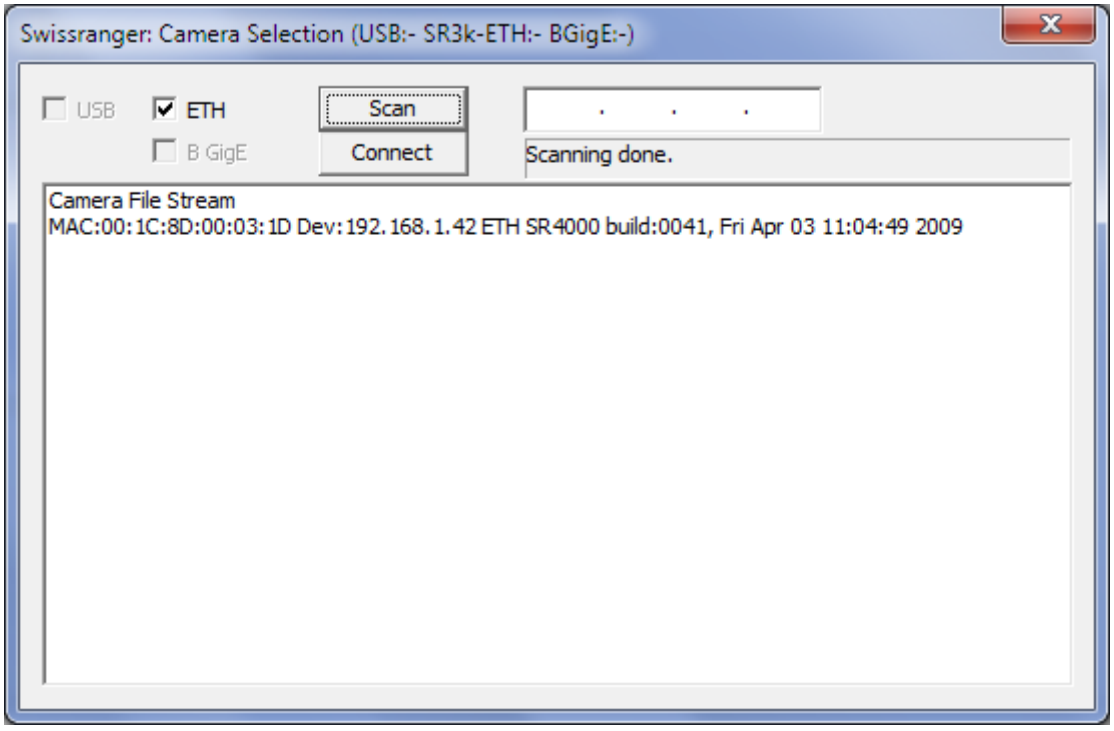


Figura 11: Cuadro de diálogo para las conexiones.

```
// this: formulario actual
SRCamera camera = new SRCamera();
IntPtr handle = new
System.Windows.Interop.WindowInteropHelper(this).Handle;
camera.ConnectDialog(handle);
```

Código 6: Ejemplo de *SRCamer.ConnectDialog ()*.

- ConnectETH (): este método realiza una conexión directa con la cámara si se conoce previamente la dirección IP de la misma que debe ser pasada por parámetro a dicho método.

  Si el método devuelve un valor negativo, indica que no se ha podido conectar con la cámara.

```
SRCamera camera = new SRCamera();
int result = camera.ConnectETH("192.168.1.42");
if (result < 0)
{
	MessageBox.Show("Error al conectar con la cámara.");
}
```

Código 7: Ejemplo de *SRCamera.ConnectETH ()*.

- ConnectUSB (): este método realiza una conexión directa con la cámara si la interfaz de comunicaciones es por medio de USB y se conoce el número de serie o ID de la cámara que se debe pasar por parámetro a dicho método. Este método no se ha usado en la aplicación del proyecto ya que la cámara no cuenta con una interfaz USB de comunicaciones.

```
SRCamera camera = new SRCamera();
int result = camera.ConnectUSB(camera.Serial);
if (result < 0)
{
	MessageBox.Show("Error al conectar con la cámara.");
}
```

Código 8: Ejemplo de *SRCamera.ConnectUSB ()*.

- Disconnect (): este método desconecta la cámara de la interfaz de comunicaciones a la cual esté conectada previamente.

```
SRCamera camera = new SRCamera();
camera.ConnectETH("192.168.1.42");
if (camera.IsConnected)
{
        camera.Disconnect();
}
```

Código 9: Ejemplo de *SRCamera.Disconnect ()*.

- GetAmplitudeThreshold (): este método obtiene el valor para el umbral del canal de amplitud. El valor para el umbral oscila entre 0 y 255.

```
SRCamera camera = new SRCamera();
int threshold = camera.GetAmplitudeThreshold();
```

Código 10: Ejemplo de *SRCamera.GetAmplitudeThreshold ()*.

- GetCartesian (): este método obtiene las coordenadas XYZ de los pixeles capturados por la cámara. El sistema natural de captado que usa la cámara es por medio de coordenadas esféricas que posteriormente deben ser transformadas en coordenadas cartesianas.

  Aunque la envolvente *SRCamAPI* proporciona tres tipos distintos de conversiones de coordenadas esféricas a coordenadas cartesianas, la clase *SRCamera* sólo acepta los tipos *float*, ya que la cámara realmente captura los pixeles en este tipo de datos básico. La palabra reservada *out* indica que los parámetros están definidos fuera y que son tratados como punteros a dichas estructuras previamente declaradas.

```
float[] xvalues, yvalues, zvalues;
SRCamera camera = new SRCamera();
int result = camera.Acquire();
if (result < 0)
{
       MessageBox.Show("Se ha producido un error en
              Acquire.");
}

camera.GetCartesian(out xvalues, out yvalues, out zvalues);
```

Código 11: Ejemplo de *SRCamera.GetCartesian ()*.

- GetCols (): este método obtiene el valor del número de pixeles que existen en la componente horizontal de la cámara. Este método es innecesario o debería ser privado ya que existe la propiedad anteriormente descrita *Width*.

```
SRCamera camera = new SRCamera();
int cols = camera.GetCols();
int cols = camera.Width;
```

Código 12: Ejemplo de *SRCamera.GetCols ()*.

- GetImage (): este método obtiene un *array* de *float* que representan un mapa o canal que ofrece la cámara determinada en base al valor de la enumeración *ImgType* que es pasado por parámetro.

  La clase *SRCamera* no tiene implementado el método *GetImageList* ya que es preferible usar el método *GetImage* y la enumeración *ImgType*.

```
float[] xvalues, yvalues, zvalues, amplitude;
SRCamera camera = new SRCamera();
camera.Acquire();
camera.GetCartesian(out xvalues, out yvalues, out zvalues);
amplitude = camera.GetImage(ImgType.IT_AMPLITUDE);
```

Código 13: Ejemplo de *SRCamera.GetImage ()*.

- GetIntegrationTime (): este método obtiene el valor del atributo interno que representa el tiempo de integración o el tiempo que consume la cámara en procesar los canales y los filtros establecidos en su configuración para cada fotograma capturado.

```
SRCamera camera = new SRCamera();
int intTime = camera.GetIntegrationTime();
```

Código 14: Ejemplo de *SRCamera.GetIntegrationTime ()*.

- GetMode (): este método obtiene el valor del atributo interno que representa el modo de captura o conjunto de filtros que se aplican al mapa de las distancias. El valor obtenido corresponde a miembros relativos a la enumeración *AcquireMode* codificado por medio de un entero.

```
SRCamera camera = new SRCamera();
int mode = camera.GetMode();
```

Código 15: Ejemplo de *SRCamera.GetMode ()*.

- GetModulationFrq (): este método obtiene el valor del atributo interno que representa la frecuencia a la cual la cámara capta cada uno de los fotogramas. Esta frecuencia está íntimamente ligada con la distancia, y dependiendo de ella se deberá usar de forma óptima una u otra frecuencia.

  El valor obtenido corresponde a un miembro de la enumeración *ModulationFrq*.

```
SRCamera camera = new SRCamera();
ModulationFrq frq = camera.GetModulationFrequency();
```

Código 16: Ejemplo de *SRCamera.GetModulationFrequency ()*.

- GetRows (): este método obtiene el valor del número de pixeles que existen en la componente vertical de la cámara. Este método es innecesario o debería ser privado ya que existe la propiedad anteriormente descrita *Height*.

```
SRCamera camera = new SRCamera();
int rows = camera.GetRows();
int rows = camera.Height;
```

Código 17: Ejemplo de *SRCamera.GetRows ()*.

- GetVersion (): este método obtiene la versión de librería *libusbsr.dll* que permite realizar la conexión con la cámara. Esta versión se puede consultar online para comprobar si existen nuevas ediciones de la librería utilizada en la aplicación aunque esta utilidad no ha sido necesaria en la implementación del proyecto.

```
SRCamera camera = new SRCamera();
try
{
	ushort[] ver = camera.GetVersions();
	Title += " [libusbsr.dll version: " +
	ver[3].ToString() + "." + ver[2].ToString() + "." +
	ver[1].ToString() + "." + ver[0].ToString() + "]";
}
catch (Exception ex)
{
	MessageBox.Show("Could not find libusbsr.dll
		driver\n(" + ex.Message + ")");
	Close();
}
```

Código 18: Ejemplo de *SRCamera.GetVersions ()*.

- OpenSettingsDlg (): este método se encarga de mostrar por pantalla el siguiente cuadro de diálogo donde se pueden modificar los dos parámetros más importantes de la cámara: *AmplitudeThreshold* y *IntegrationTime*.

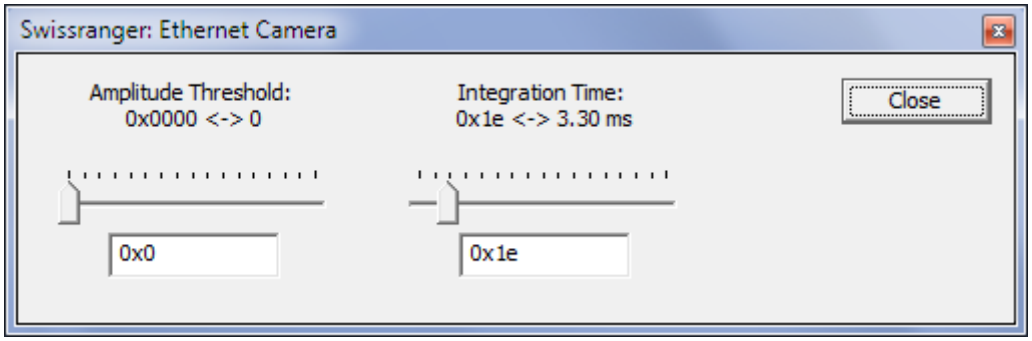


Figura 12: Cuadro de diálogo de los parámetros más importantes.

```
SRCamera camera = new SRCamera();
IntPtr handle = new
System.Windows.Interop.WindowInteropHelper(this).Handle;
camera.ConnectETH("192.168.1.42");
if (camera.IsConnected)
{
        camera.OpenSettingsDlg(handle);
}
```

Código 19: Ejemplo de *SRCamera.OpenSettingsDlg ().*

- ReadSerial (): este método obtiene el número de serie o ID de la cámara que posea una interfaz de comunicaciones basada en la tecnología USB. La cámara que se ha utilizado para el desarrollo del proyecto carece de dicha interfaz.

  Devuelve un valor entero que representa dicho número de serie. Este método es innecesario o debería ser privado ya que existe la propiedad anteriormente descrita *Serial*.

```
SRCamera camera = new SRCamera();
int serial = camera.ReadSerial();
```

Código 20: Ejemplo de *SRCamera.ReadSerial ().*

- SetAmplitudeThreshold (): este método asigna un valor pasado por parámetro al atributo interno que representa el umbral para el canal de las distancias.

En el ejemplo siguiente se pasa por parámetro el valor 528 que ha sido el valor óptimo para el umbral usado durante el desarrollo del proyecto.

```
SRCamera camera = new SRCamera();
camera.SetAmplitudeThreshold(528);
```

Código 21: Ejemplo de *SRCamera.SetAmplitudeThreshold ().*

- SetIntegrationTime (): este método asigna un valor pasado por parámetro al atributo interno que representa el tiempo de integración o tiempo que la cámara consume en procesar los canales y de aplicar los filtros necesarios.

  En el ejemplo siguiente se pasa por parámetro el valor 3 que ha sido el valor óptimo para el tiempo de integración durante el desarrollo del proyecto.

```
SRCamera camera = new SRCamera();
camera.SetIntegrationTime(3);
```

Código 22: Ejemplo de *SRCamera.SetIntegrationTime ().*

- SetMode (): este método asigna un valor pasado por parámetro al atributo interno que representa los filtros que la cámara deberá aplicar al mapa de las distancias. El valor pasado puede corresponder a uno o varios miembros de la enumeración *AcquireMode*.

  Aunque los valores por defecto ya se han descrito anteriormente, para el desarrollo de la aplicación se ha preferido optar por los siguientes filtros que resultan más óptimos además de poder presentar un número mayor de ellos.

```
SRCamera camera = new SRCamera();
camera.ConnectETH("192.168.1.42");
if (!camera.IsConnected)
{
        MessageBox.Show("Could not connect to camera");
}
else
{
        camera.SetMode(AcquireMode.AM_COR_FIX_PTRN |
        AcquireMode.AM_DENOISE_ANF | AcquireMode.AM_CONF_MAP
        | AcquireMode.AM_MEDIAN);
}
```

Código 23: Ejemplo de *SRCamera.SetMode ()*.

-  SetModulationFrq (): este método asigna un valor pasado por parámetro al atributo interno que representa la frecuencia de captación de fotogramas de la cámara. El valor pasado corresponde a un miembro de la enumeración *ModulationFrq*.

  En el siguiente ejemplo de frecuencia válida tanto para los modelos SR3000 como para los modelos SR4000.

```
SRCamera camera = new SRCamera();
cam.SetModulationFrequency(ModulationFrq.MF_30MHz);
```

Código 24: Ejemplo de *SRCamera.SetModulationFrequency ()*.

-  SRCamera (): este método representa el constructor de la clase *SRCamera* y está sobrecargado permitiendo el paso de un parámetro que corresponde al puntero manejador de la cámara que se pretende crear.

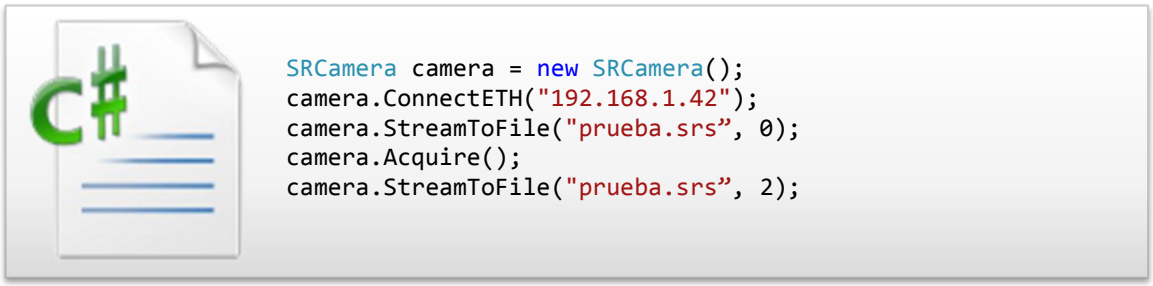
```
IntPtr handle = new IntPtr();
SRCamera camera1 = new SRCamera(handle);
SRCamera camera2 = new SRCamera();
```

Código 25: Ejemplo de constructores de *SRCamera*.

-  StreamToFile (): esté método se encarga de manejador un fichero que contiene fotogramas almacenados grabados de una sesión anterior por parte de la cámara. Para ello requiere de un nombre de fichero donde se alojarán los fotogramas y un modo de operación para el tratamiento del fichero. Los modos de operación son:

    - 0: para abrir un fichero sino existe y si existe, truncarlo.
    - 1: para abrir un fichero sino existe y si existe, añadir al final del fichero (no implementado por el fabricante).
    - 2: para cerrar el fichero.

    En el siguiente ejemplo se abre el fichero *prueba.srs* (*srs = SRStream*), se captura un fotograma por medio de la cámara y se cierra el fichero.



```
SRCamera camera = new SRCamera();
camera.ConnectETH("192.168.1.42");
camera.StreamToFile("prueba.srs”, 0);
camera.Acquire();
camera.StreamToFile("prueba.srs”, 2);
```

Código 26: Ejemplo de *SRCamera.StreamToFile ()*.

Capítulo 3

Interfaz Gráfica

3.1 Claves

La claves para este capítulo son:

- Describir la tecnología WPF.
- Describir NatalSoft.
- Introducir el volante virtual.

3.2 Introducción

En este capítulo se desglosa la interfaz gráfica creada para poder utilizar la cámara más cómodamente observando los canales y filtros mediantes imágenes en tiempo real. Uno de los objetivos de la interfaz gráfica y de este apartado del proyecto es demostrar que la tecnología .NET puede ser sobradamente necesaria para acometer con tales tareas sin necesidad de introducirse en lenguajes más costosos, aunque más eficientes, como pueda ser C++ no manejado combinado con las librerías *OpenCV* de Intel.

Otro aspecto a tener en cuenta, es la integración de multitud de controles y librerías desarrollados por terceros que apenas han supuesto coste alguno en su manipulación e incorporación al proyecto. La reducción del impacto de usar librerías en .NET, sobre todo en el ahorro temporal, ha supuesto poder cumplir con mayor funcionalidad de la inicialmente planificada.

Por tanto, se han impuesto dos objetivos que bien podrían ser objeto de largas discusiones, pero por parte del desarrollador del proyecto se han visto necesariamente impulsados:

- buscar la eficacia y no la eficiencia usando lenguajes más cómodos en pro de una mayor funcionalidad a corto plazo.
- primar la interoperabilidad entre multitud de componentes para consolidar logros autoimpuestos.

3.3 WPF, XAML y MVVM

Antes de pasar a describir la interfaz gráfica es necesario detallar la tecnología utilizada en el proyecto para diseñar la GUI. La tecnología empleada supone un cambio brusco y radical en la forma en la cual se programan y gestionan las interfaces de usuario.

Se ha optado por una tecnología sencilla de implementar y que ofrece multitud de técnicas y librerías que aumentan la interoperabilidad de la aplicación sin coste alguno para el proyecto.

Para lograr dichos objetivos se ha hecho uso del conjunto de librerías de representación denominado WPF (*Windows Presentation Foundation*) que ofrece mejoras sustanciales en cuanto al anterior conjunto de librerías.

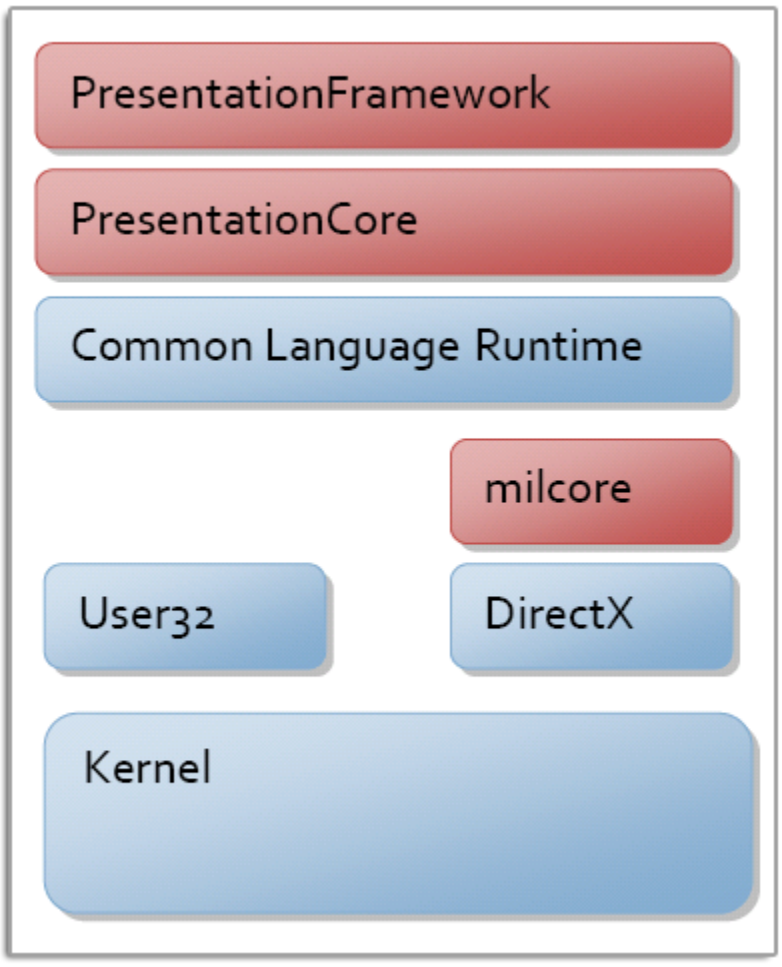


Figura 13: Arquitectura de Windows Presentation Foundation.

WPF es un subsistema gráfico para el renderizado de interfaces de usuario para aplicaciones basadas en Windows. En lugar de basarse en GDI, que suponía un alto coste

computacional por parte del procesador, todos los elementos gráficos se ejecutan en *DirectX* obteniéndose mejores resultados, incluso ofreciendo la posibilidad de introducir elementos tridimensionales. Las capas presentadas en la imagen de color rojo indican que representan la mayor parte del subsistema WPF. Este subsistema está dividido en las siguientes capas:

- PresentationFramework: representa la capa visible del *framework* de presentación de gráficos.
- PresentationCore: representa el núcleo o la capa de aceptación de peticiones para ser procesadas.
- Common Language Runtime: representa el núcleo de .NET por la cual todas las peticiones de la máquina virtual son procesadas.
- MilCore: es la única capa en rojo cuyo código no está administrado para poder mantener la interoperabilidad con la capa inferior *DirectX*.
- User32: capa de presentación GDI para los controles antiguos. Esta capa se mantiene para poder integrar distintas versiones de Microsoft Windows.
- DirectX: capa de presentación de datos que está en contacto con la tarjeta gráfica.
- Kernel: capa del sistema operativo por la cual pasan las llamadas desde la capa *User32* o *DirectX*.

Para realizar esta tarea WPF utiliza un nuevo lenguaje derivado de XML denominado XAML. Las aplicaciones WPF pueden ser usadas tanto como programas de escritorio o como páginas web. El objetivo de dicho pilar o subsistema es la unificación de todos los elementos gráficos visibles por el usuario tales como tipografías, documentos, textos, dibujos bidimensionales y tridimensionales, luces, escenas, cámaras, vídeos, audios, etc.


```
<Canvas xmlns="http://schemas.microsoft.com/client/2007"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
       <TextBlock>
              Hello World!
       </TextBlock>
</Canvas>
```

Código 27: Ejemplo de código XAML.

Para otorgar a esta nueva generación de aplicaciones de escritorio y web un marcado carácter ingenieril, todo el subsistema gráfico explicado y consecuentemente las aplicaciones desarrolladas bajo él, se basan en un nuevo patrón arquitectónico denominado MVVM (*Model-View-ViewModel*). Para muchos desarrolladores de software dicho patrón no lo es como tal o bien es un plagio readaptado de Microsoft al patrón arquitectónico Model-Vista-Controlador. Sin embargo, dicho patrón cumple varias funcionalidades muy importantes a la hora de gestionar un proyecto software moderno y adaptado a las necesidades de hoy que ofrecen las interfaces gráficas.

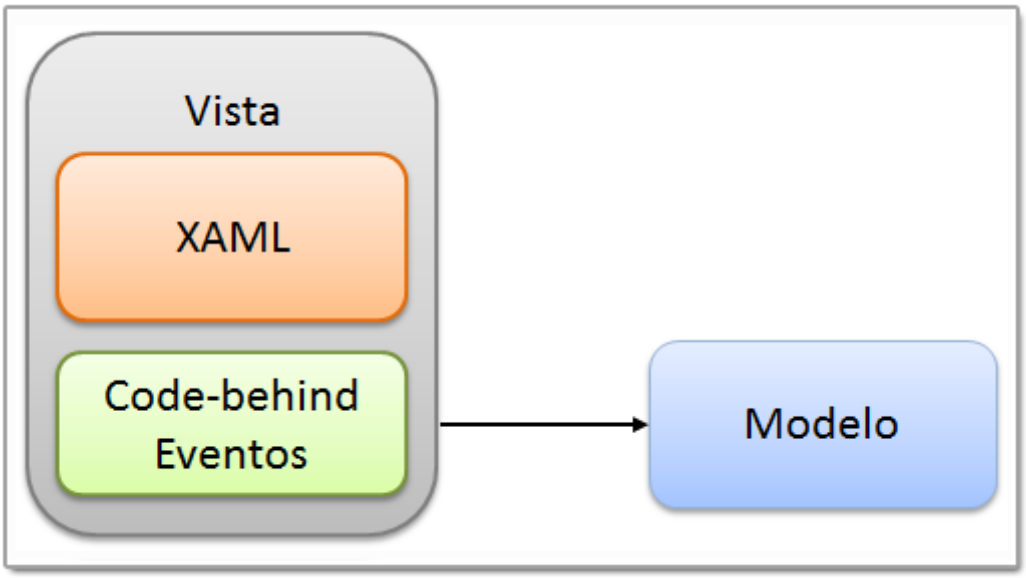


Figura 14: Patrón Modelo-Vista-Controlador.

Las ventajas introducidas por el patrón MVVM son varias pero las más importantes son:

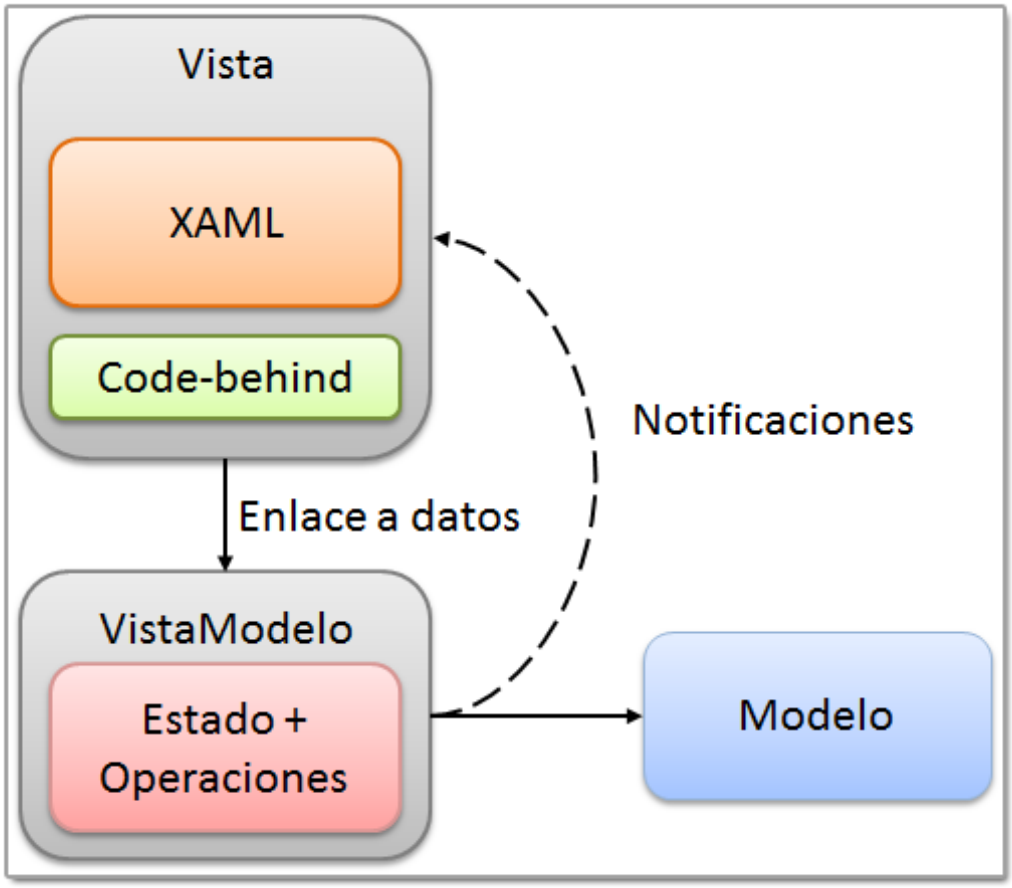


Figura 15: Patrón Modelo-Vista-VistaModelo.

En cuanto a la capa Vista:

- Para desarrollar la interfaz gráfica se requiere de XAML pero no se requiere de código programado. Es importante destacar que XAML define la interfaz gráfica y su comportamiento, pero no la lógica del código (flujos condicionales o bucles).
- La interfaz gráfica se encuentra separada en ficheros propios a los cuales el equipo de diseñadores puede gestionar. Es importante a la hora de paralelizar el grupo de trabajo y delegar las funciones de diseño a aquellos diseñadores entrenados para dicha tarea.

En cuanto a la capa VistaModelo:

- Se ha creado una interfaz sólida y consistente de comunicaciones entre la capa Vista y la capa VistaModelo para poder interactuar con los objetos gráficos sin necesidad de introducir código en la capa Modelo. Los controles y las acciones gráficas permanecen en un contexto gráfico donde los diseñadores pueden realizar su trabajo. A esta interfaz de comunicaciones se le denominada Enlace a Datos (*Data Binding*) y se explica más adelante.
- Se pueden realizar clases para testear dichas comunicaciones lo que aumenta el grado de robustez de las aplicaciones.

En cuanto a la capa Modelo:

- Se pueden realizar clases para testear las clases del modelo lo que aumenta el grado de robustez de las aplicaciones.

3.3.1 Data Binding

En este apartado se presenta lo que aporta la tecnología *Data Binding* a las aplicaciones en WPF.

Como se ha mencionado anteriormente, representa una interfaz de comunicaciones o *enlace a datos* por medio del patrón *Observer* en su versión más moderna: *el evento* entre dos propiedades que forman parte de la definición de las dos clases involucradas. Cuando el usuario realiza operaciones sobre los controles visuales de las aplicaciones, dichas acciones se traducen a eventos que generan peticiones de actualización del estado gráfico.

Este sistema mantiene la aplicación en un estado persistente sin que el programador, desarrollador o diseñador tengan que realizar tareas más que la propia integración del enlace a los datos.

Los enlaces a datos pueden tener direcciones en cuanto a los objetos que requieren de la actualización, habiendo tres formas posibles de actualización:

- Directa (*OneWay*): el objeto destino es actualizado por los datos que se encuentran en el objeto origen.
- Inversa (*OneWayToSource*): el objeto origen es actualizado por los datos que se encuentran en el objeto destino.

- Completa (*TwoWay*): tanto el origen como el destino mantienen el estado persistente de los datos en una única transacción de comunicaciones.

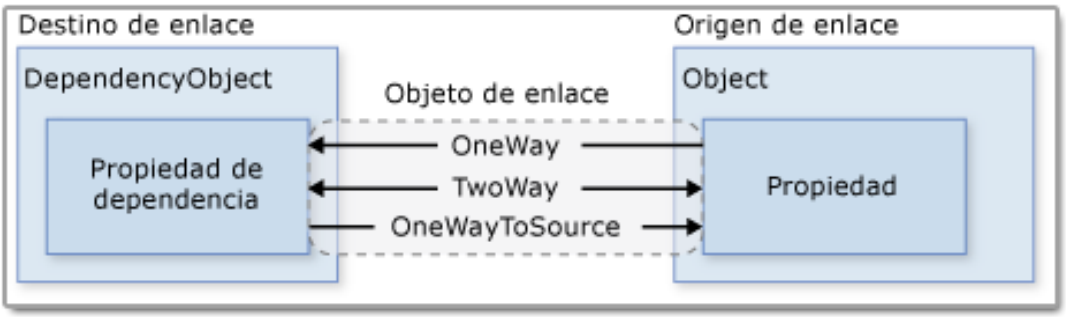


Figura 16: Direcciones del Enlace a Datos.

La funcionalidad básica que ofrece los enlaces a datos son los siguientes:

- Conversión de datos en la capa gráfica (*View*) para mostrar los datos origen en la interfaz gráfica de una manera distinta.
- Enlace a colecciones de datos en orígenes diversos como bases de datos, ficheros planos, ficheros Excel bajo un entorno unificado.
- Inclusión de datos en plantillas para su reutilización y abstracción de modelos gráficos.
- Validación de datos previa a la realización de acciones por parte del usuario.
- Mecanismo de depuración para poder recibir el estado de las comunicaciones del enlace de datos.

3.4 NatalSoft

En esta sección del capítulo se muestra la aplicación desarrollada para la ejecución del proyecto que hace uso de la librería *SwissRanger.dll* ya explicada conjuntamente con la cámara SR4000. A continuación se presenta una captura de pantalla de la aplicación en tiempo de ejecución.

La aplicación ha servido en primer lugar para crear la API *SwissRanger* bajo la arquitectura .NET para futuras creaciones de otras aplicaciones, y en segundo lugar, para crear una clase que abstrae virtualmente un volante de un coche. Esta segunda funcionalidad de la aplicación y del proyecto se explica con más detalle en la prueba de concepto realizada dentro del Capítulo 4: Aplicación de cámaras TOF al ocio.

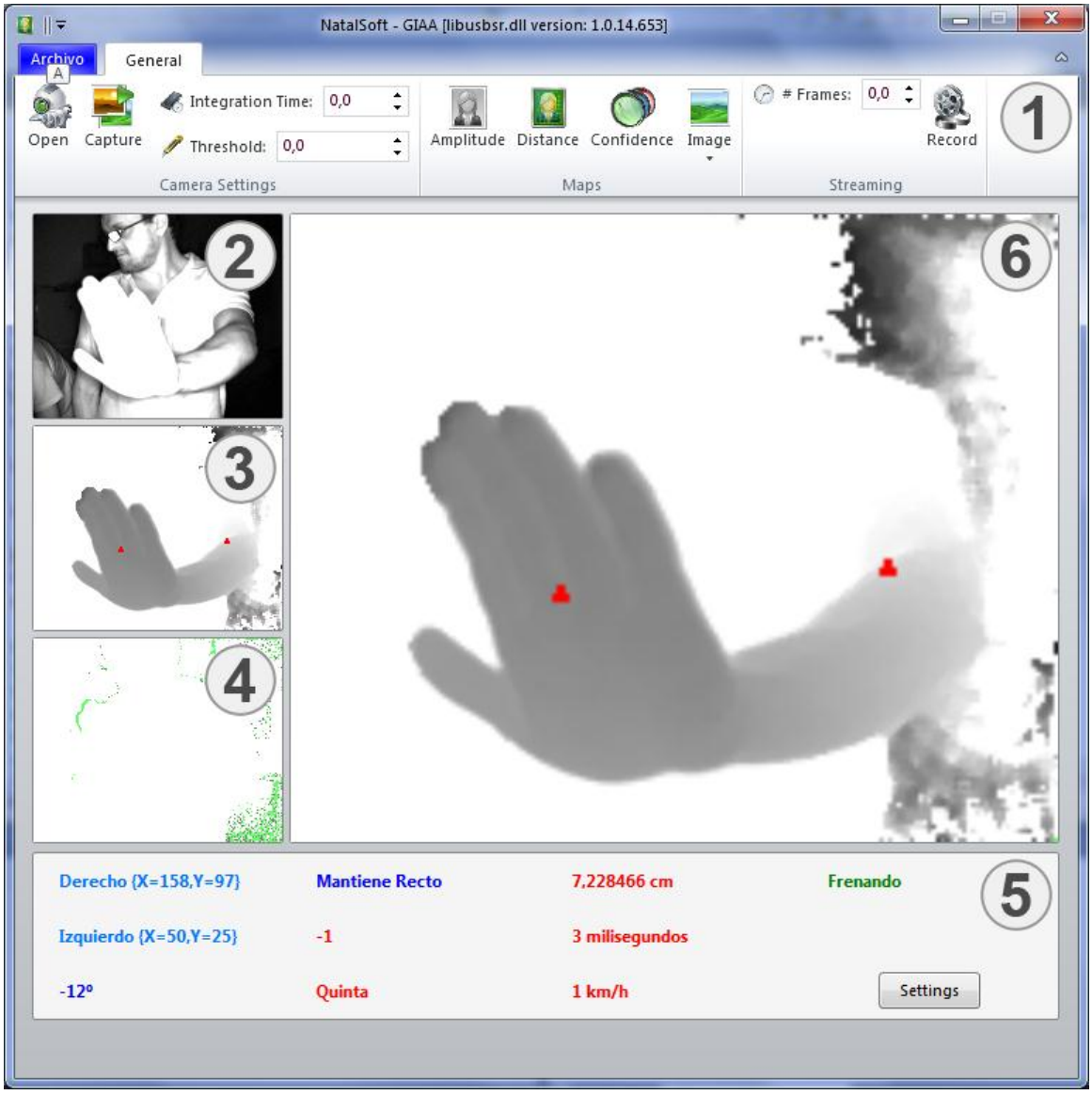


Figura 17: Captura de NatalSoft.

Como se puede observar en la captura de pantalla, la aplicación está dividida en seis partes que ofrecen una funcionalidad específica al usuario. Las seis partes que constituyen la aplicación son:

- 1 Barra de menú.
- 2 Mapa de amplitud.
- 3 Mapa de distancia.
- 4 Mapa de confianza.
- 5 Cuadro de mando del volante virtual.
- 6 Zoom a uno de los mapas.

3.4.1 Barra de menú

La barra de menú ha sido implementada siguiendo el estándar para barras de menús usadas en la aplicación *Microsoft Office 2010* y que empezó en *Microsoft Office 2007*. Debido a que dicha forma de exponer los elementos de la funcionalidad requerida por la aplicación son acciones realizables por el usuario, esta barra de menú ayuda al entendimiento y comprensión de tales acciones.

Para poder usar esta barra de menú en cualquier aplicación se requiere de la librería gratuita *Fluent Ribbon Control Suite* suministrada por CodePlex. Para ello, el formulario WPF creado debe simplemente heredar del objeto *Fluent* existente en la librería.

Un ejemplo de código XAML que ayudará a entender la programación de la barra:

```
<Fluent:RibbonWindow x:Class="NatalSoft.MainWindow"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       xmlns:Fluent="clr-namespace:Fluent;assembly=Fluent"
       xmlns:Main="clr-namespace:NatalSoft">
</Fluent:RibbonWindow>
```

Código 28: Uso del ensamblado *Fluent* en la interfaz gráfica.

En este ejemplo de código la clase definida en el proyecto denominada *MainWindow* hereda del comportamiento del objeto *RibbonWindow* definido en el ensamblado *Fluent*. Este ensamblado ha sido importado en la aplicación por la línea de código *xmlns:Fluent* donde se ha especificado la ruta de la referencia del proyecto.

A su vez la barra de menú se encuentra subdividida en tres áreas o grupos de acciones que separan la funcionalidad de la aplicación. Las áreas o grupos son los siguientes:

- Camera Settings.
- Maps.
- Streaming.

El área o grupo de acciones que se engloban en *Camera Settings* está asociado a la forma en la cual la cámara capta las imágenes que luego se mostrarán por pantalla. Las acciones que el usuario puede realizar son:

- Open: abre la cámara para establecer conexión con ella y poder capturar los fotogramas captados. Para establecer la conexión con la cámara se muestra el cuadro de diálogo de la Figura 11 donde se da la opción de elegir unas de las cámaras que estén conectadas a la interfaz de comunicaciones o bien poder conectar con la cámara virtual para observar fotogramas grabados en anteriores sesiones.
- Capture: una vez la cámara abierta y conectada con la interfaz, se puede proceder a la captura de los fotogramas por parte de la misma. Esta acción inicia un hilo de ejecución que cada un milisegundo ejecuta la rutina para capturar el fotograma de la cámara.
- Integration Time: uno de los parámetros más importantes de la cámara, que determina el tiempo empleado por la cámara para el procesado de los canales y de los filtros.
- Threshold: el segundo parámetro más importante de la cámara que establece qué pixeles se muestran en los canales y cuales no en función de la distancia a la cámara.

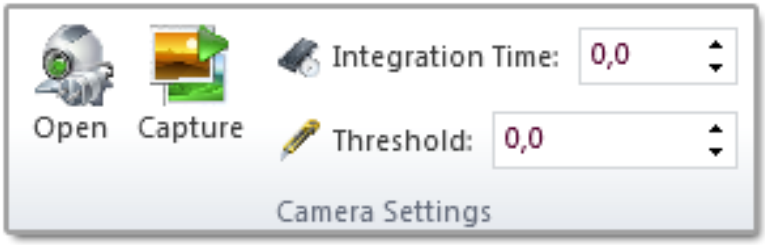


Figura 18: Área *Camera Settings*.

El área o grupo de acciones que se engloban en *Maps* está asociado a la forma en la cual se presentan por pantalla los fotogramas captados en base a los canales y filtros preestablecidos por la aplicación. Las acciones que el usuario puede realizar son:

- Amplitude: determina si la cámara debe realizar la captura de los fotogramas por el canal de amplitud, y a su vez si se debe mostrar dicho canal por la interfaz gráfica.
- Distance: determina si la cámara debe realizar la captura de los fotogramas por el canal de distancias, y a su vez si se debe mostrar dicho canal por la interfaz gráfica.

- Confidence: determina si la cámara debe realizar la captura de los fotogramas por el canal de confianza, y a su vez si se debe mostrar dicho canal por la interfaz gráfica.
- Image: determina si se muestra en el visor central de la aplicación uno de los tres canales anteriormente citados o si no se muestra ninguno.

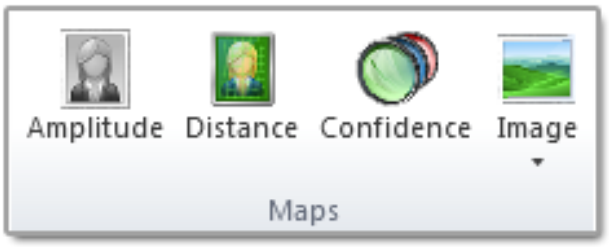


Figura 19: Área *Maps*.

La Figura 12 muestra la galería de canales que se muestran en la imagen central de la aplicación.

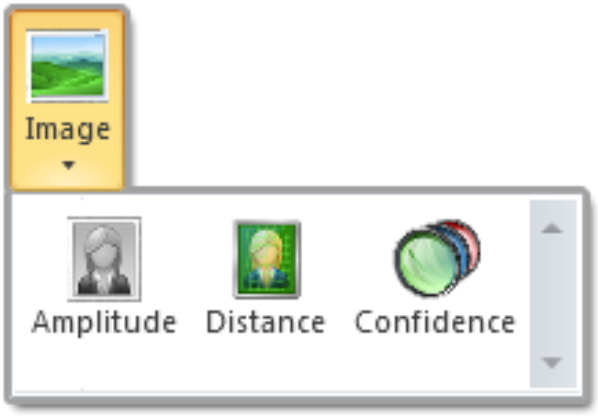


Figura 20: Galería de canales.

A continuación se muestra un extracto de código de cómo se ha creado dicho objeto que contiene a los tres botones para mostrar una de ellas en la imagen central.

```
<Fluent:DropDownButton x:Name="cmbBigImage" Text="Image"
       LargeIcon="Images/Favorite-64.png" >
       <Fluent:Gallery>
              <Fluent:ToggleButton x:Name="btnBigAmplitude"
                     Text="Amplitude"
                     LargeIcon="Images/Favorite-64.png"
                     Checked="btnBigAmplitude_Checked"/>
              <Fluent:ToggleButton x:Name="btnBigDistance"
                     Text="Distance"
                     LargeIcon="Images/Favorite-64.png"
                     Checked="btnBigDistance_Checked"/>
              <Fluent:ToggleButton x:Name="btnBigConfidence"
                     Text="Confidence"
                     LargeIcon="Images/Favorite-64.png"
                     Checked="btnBigConfidence_Checked"/>
       </Fluent:Gallery>
</Fluent:DropDownButton>
```

Código 29: Creación de un objeto *DropDownButton*.

El área o grupo de acciones que se engloban en *Streaming* está asociado a la forma en la cual se graban los fotogramas captados por la cámara para posteriormente ser visualizados por la cámara virtual por medio del cuadro de diálogo expuesto en la Figura **11**. Las acciones que el usuario puede realizar son:

- Record: determina el comienzo de la grabación de los fotogramas captados por la cámara. Si el objeto que determina los fotogramas máximos de grabación estuviese a cero, se grabaría el máximo de fotogramas permitidos por la aplicación para no crear un fichero de salida excesivamente grande, que puede llegar a ocupar gigas de espacio en disco.
- Number of Frames: determina el número de fotogramas que serán grabados en el fichero destino.

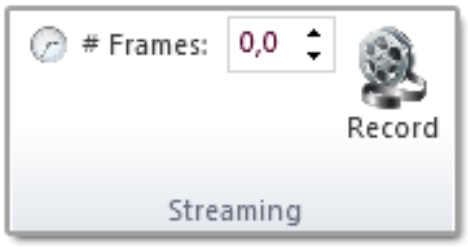


Figura 21: Área *Streaming*.

3.4.2 Mapa de amplitud

El mapa de amplitud hace referencia a una de las entradas de la enumeración *ImgType* ya explicada en el capítulo anterior. El campo dentro de la enumeración se denomina IT_AMPLITUDE. Este canal actúa como la grabación de una cámara normal con representación de la imagen en escala de grises.

Este canal de información no aporta inteligencia a la aplicación, sino que ayuda a mantener la coherencia entre los demás canales de información.

3.4.3 Mapa de distancia

El mapa de distancias hace referencia a una de las entradas de la enumeración *ImgType* ya explicada en el capítulo anterior. El campo dentro de la enumeración se denomina IT_DISTANCE. Este canal almacena la información de los pixeles captados en función de la distancia que tienen los objetos de la escena a la cámara.

Como puede observarse también está representado en escala de grises. Los elementos oscuros determinan la proximidad a la cámara mientras que los pixeles en blanco determinan la lejanía, estableciéndose así la escala de distancias al punto de grabación.

Si en el mapa se dibujasen pixeles en verde, esto indicaría que los pixeles pertenecen al plano infinito ya que la cámara posee una distancia óptima de grabación. Si esta distancia óptima es superada por un objeto, dicho objeto no aparecerá en el canal, y dejará una zona de lejanía máxima o infinita.

Este canal presenta además los centroides de los puños detectados si una persona está presente en la escena.

3.4.4 Mapa de confianza

El mapa de confianza hace referencia a una de las entradas de la enumeración *ImgType* ya explicada en el capítulo anterior. El campo dentro de la enumeración se denomina IT_CONF_MAP. Este canal, aunque no es usado en el desarrollo de más aplicaciones ni en la interpretación o ayuda de otros canales, determina un papel complementario a los demás canales de información.

Los pixeles que aparecen en blanco determinan la confianza o grado de certidumbre que presenta dicho pixel. Un pixel en verde no presenta confianza ya que es captado más allá de la distancia óptima (infinito) y puede conllevar a interpretaciones erróneas de la situación real de la escena.

Este canal, dependiendo del uso, se podrá usar como tal o bien como un filtro para la imagen real, y para su posterior procesado. La cámara no puede presentar dicho canal como filtro dentro del conjunto de filtros, ya que esto supondría un mayor coste en tiempo para el procesado del canal.

Es por eso que se presenta como canal, así el usuario posee un grado superior de control sobre la certeza o incertidumbre que presentan los pixeles de la escena, pudiendo aplicar otras técnicas computacionales más complejas.

3.4.5 Cuadro de mandos

Uno de los principales objetivos del proyecto es formalizar un conjunto de librerías para interactuar de forma más cómoda con la cámara, y otro de ellos es realizar una prueba de concepto basada en la utilización de dicha librería.

El volante virtual es el producto de haber realizado un uso intensivo de la librería que maneja la cámara tridimensional. La definición de la clase y su comportamiento serán explicados en el capítulo siguiente con más detalle.

Por el momento sólo se describirá el conjunto de campos o atributos que aparecen en la interfaz gráfica y que sirven para simular un volente de un vehículo más algunas propiedades de un motor de conducción manual.

Las propiedades del volante virtual son:

- Centroide izquierdo: determina la posición en la que se encuentra la mano izquierda del conductor del vehículo.
- Centroide derecho: determina la posición en la que se encuentra la mano derecha del conductor del vehículo.
- Pendiente: determina la recta tangente o pendiente que pasa por los centroides izquierdo y derecho. Sirve principalmente para calcular el ángulo de giro y con ello, la dirección de giro o intención de giro del conductor del vehículo.
- Dirección de giro: campo que usa la pendiente para presentar al conductor del vehículo la intención de giro. Es importante denotar que una pendiente cero raras veces se da, por tanto se ha introducido un umbral en el cual el volante se encuentra recto aunque la pendiente sea distinta de cero.
- Aceleración: indica la aceleración que está sufriendo el vehículo en función de variables como la masa, la fricción y otros parámetros físicos.
- Marcha: indica la marcha en la que se encuentra el motor. Las condiciones para aumentar o disminuir de marcha son gestos que el conductor del volante debe realizar con las manos.
- Distancia: es la distancia en centímetros que existen entre los dos centroides. La principal funcionalidad que tiene el calcular la distancia es la clasificación de la conducta del usuario sobre el vehículo. La conducta en función de la distancia puede ser frenar o acelerar. Dependiendo del valor número de la distancia euclídea obtenida, se puede establecer un grado de intensidad en las acciones.
  Un ejemplo de ello es, a más distancia más aceleración, cuando se está acelerando, o a menos distancia, mayor frenada cuando se está frenando.
- Milisegundos: el número de milisegundos que han pasado sin que se produzca una acción determinante para el volante. Para que las acciones que el conductor realiza sobre el volante se consideren para ser ejecutadas, debe pasar un tiempo impuesto por el programador donde la acción es siempre la misma. Si la acción cambia, el número de milisegundos se inicializa a cero y debe volver a ejecutarse la acción.
- Velocidad: la velocidad que lleva el vehículo en función de la aceleración y de la marcha en la cual se encuentra el motor.
- Movimiento: campo que usa la distancia entre los centroides para detectar si el conductor del vehículo requiere que el motor acelere o que el motor frene. El cuadro de los movimientos previsibles que el usuario puede realizar y el diagrama de estados se presenta en el capítulo siguiente.
- Botón Settings: este botón abre un cuadro de diálogo interno a la cámara que permite la modificación de las dos variables más importantes de la cámara: el tiempo de integración y el umbral para la distancia. El cuadro que aparece se muestra en la Figura 12.

3.4.6 Imagen central

La imagen central permite ver uno de los tres canales seleccionados por el usuario de forma aumentada. Para ello se hace referencia al canal seleccionado y se adapta la imagen a las dimensiones del contenedor, pero no por ello, la resolución de la imagen se ve aumentada.

Capítulo 4

Aplicación de cámaras TOF al ocio

4.1 Claves

La claves para este capítulo son:

- Describir la tecnología XNA.
- Describir el volante virtual.
- Describir la prueba de concepto.

4.2 Introducción

En este capítulo se presenta otro conjunto de librerías que permiten a los desarrolladores crear videojuegos orientados a objetos bajo una máquina virtual que administre la memoria. Esta librería ha sido desarrollada por Microsoft como futura plataforma de desarrollo para los videojuegos producidos para su consola XBOX. Actualmente Microsoft ha dejado de mantener la plataforma de desarrollo *DirectX Managed* en pro de XNA, que es la librería objeto de este capítulo.

Una vez descrita y enseñada la tecnología utilizada, se expone la funcionalidad desarrollada con dicha tecnología. El elemento desarrollado ha sido un juego en tres dimensiones, donde el volante virtual introducido en el capítulo anterior forma parte del engranaje del juego.

Con esta funcionalidad, se pretende acercar las cámaras TOF al ocio digital como ya está haciendo Microsoft con su proyecto Natal y su producto recientemente presentado *Kinect*.

4.3 Arquitectura XNA

En esta sección se describe la arquitectura XNA para el desarrollo de videojuegos en C# bajo la infraestructura de .NET. Los juegos desarrollados pueden ejecutarse en XBOX, PC, ZUNE o Windows Mobile, sin realizar adaptaciones en el código, lo que convierte a XNA en un modelo fácil y cómodo para crear videojuegos.

El lenguaje de programación escogido es C#, que es un lenguaje de programación orientado a objetos bajo la máquina virtual de .NET.

A continuación se muestra la línea de trabajo o ejecución que sigue una aplicación desarrollada en XNA.

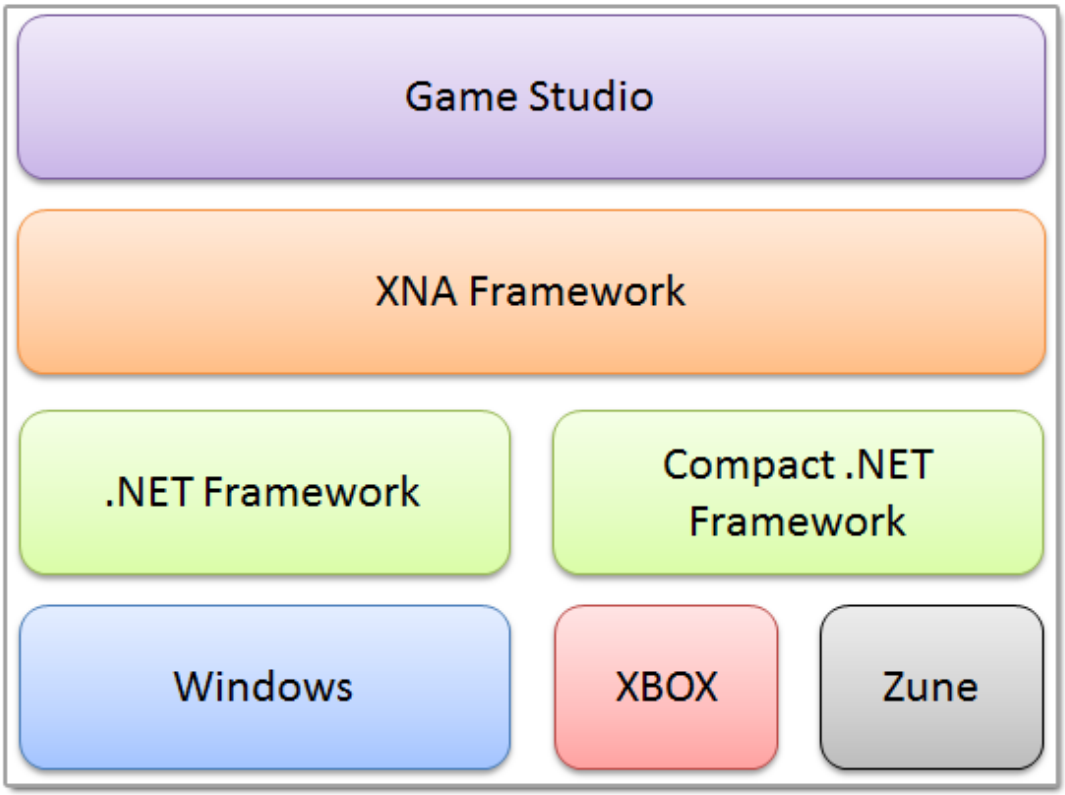


Figura 22: Arquitectura de XNA.

La primera capa de este modelo atiende a la creación del código en un entorno de programación (IDE). Este entorno de programación suele ser Microsoft Visual Studio 2008 en su versión Express o recientemente su versión mejorada Microsoft Visual Studio 2010, donde permite la creación de videojuegos para dispositivos móviles.

En la segunda capa del modelo se presenta el conjunto de librerías de XNA. Estas librerías contienen ensamblados directamente ejecutables desde *DirectX* lo que convierte a XNA en un conjunto de librerías muy potente. Cuando se realiza un juego, se debe respetar el modo de funcionamiento interno de ejecución. Este modelo interno de

ejecución se denomina XNA Looping y consiste básicamente en un bucle infinito que ejecuta dos métodos básicos sobrescritos por todos los objetos XNA: *Update()* y *Draw()*. En la imagen siguiente se muestra el flujo de ejecución de un objeto XNA.

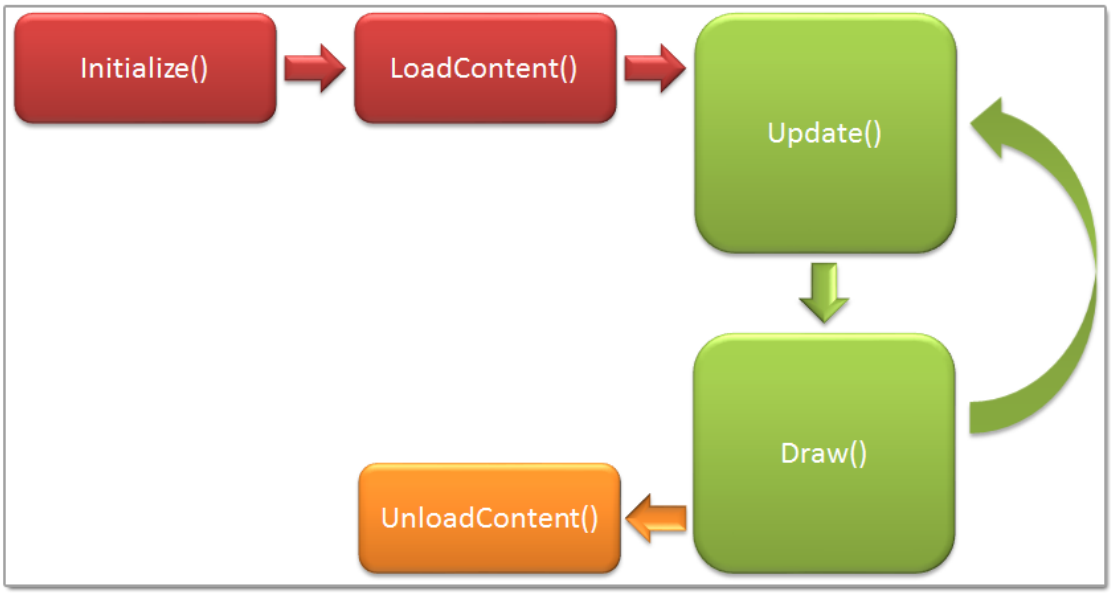


Figura 23: XNA Looping.

El objeto comienza con un método que inicializa las variables de estado en función del contexto del objeto. Una vez concluida dicha función, se pasa a cargar los contenidos que el objeto XNA tiene establecidos. Los contenidos que un objeto XNA puede contener son múltiples como texturas, sonidos, elementos tridimensionales, o conjunto de ellos. La función de carga de contenido requiere de una complejidad computacional elevada principalmente por la conversión de modelos tridimensionales en ficheros que deben ser portados a memoria. Esta complejidad se vuelve mayor cuando se requiere de un mapeo propio de los objetos tridimensionales importados con texturas diseñadas en entornos ajenos.

Los métodos de XNA Looping (en verde) se ejecutan en un bucle mantenido por el entorno de ejecución de .NET. El método *Update()* sirve para actualizar el estado de las instancias de objetos XNA, mientras que *Draw()* se encarga de renderizar el objeto XNA en pantalla una vez actualizado su estado. Todos los objetos XNA que deban ser renderizados deberán heredar de una interfaz o clase común y posteriormente reescribir los dos métodos de XNA Looping.

En la tercera capa de la arquitectura XNA se dispone de las librerías .NET dependiendo del destino de ejecución. Si el videojuego desarrollado se ejecuta en Windows, el conjunto de librerías en el cual delega la ejecución es la máquina virtual de .NET con toda la funcionalidad. Sin embargo, se ha creado un conjunto de librerías de reducido tamaño denominado Compact .NET Framework para que los dispositivos móviles y la XBOX puedan ejecutar el código de los videojuegos desarrollados en XNA.

En la última capa se presentan los distintos entornos de ejecución en los cuales se pueden disfrutar de videojuegos desarrollados con XNA. Actualmente XNA se encuentra en su cuarta versión siendo más estable y mucho más rápida su ejecución lo que ha

permitido que dispositivos móviles con Windows Mobile puedan ejecutar los juegos creados.

4.4 RacingGame

En esta sección del documento se explica cada uno de los namespaces que componente el proyecto de ejecución del videojuego RacingGame. Este proyecto forma parte de los kits de iniciación a XNA proporcionados por el club de creadores de XNA.

Aunque el proyecto se realizó inicialmente para XNA 1.0, se ha mantenido hasta su versión 3.1 de XNA, añadiéndose en cada versión las mejoras que a su vez incluían las librerías como *Shaders*, ejecución ágil de elementos tridimensionales o actualizaciones en XACT.

La solución de la aplicación consta de dos proyectos:

- RacingGame: este proyecto representa el 99% de la aplicación y es objeto de explicación en detalle de esta sección del documento
- RacingGameContentProcessors: este proyecto sirve para cargar en memoria los modelos tridimensionales almacenados en ficheros en la carpeta *Content* del proyecto *RacingGame*. Este proyecto sólo contiene una clase que se encarga de gestionar el modelo tridimensional en memoria y de aplicar los mapas de texturas que este modelo requiera.

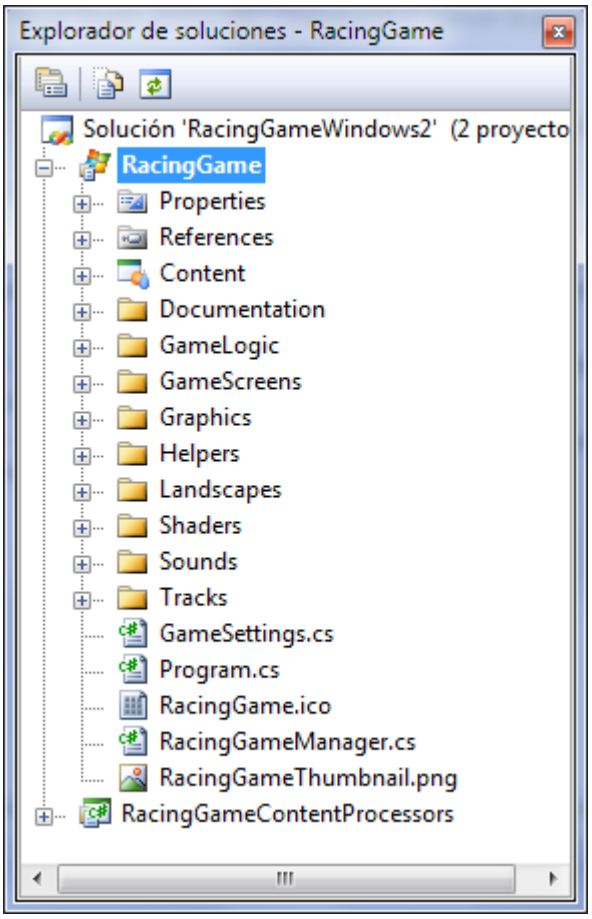


Figura 24: Proyecto RacingGame.

Aunque el proyecto *RacingGame* consta de muchos namespaces sólo se describen los de nivel superior para ofrecer una visión de alto nivel del videojuego. Los namespaces más importantes son:

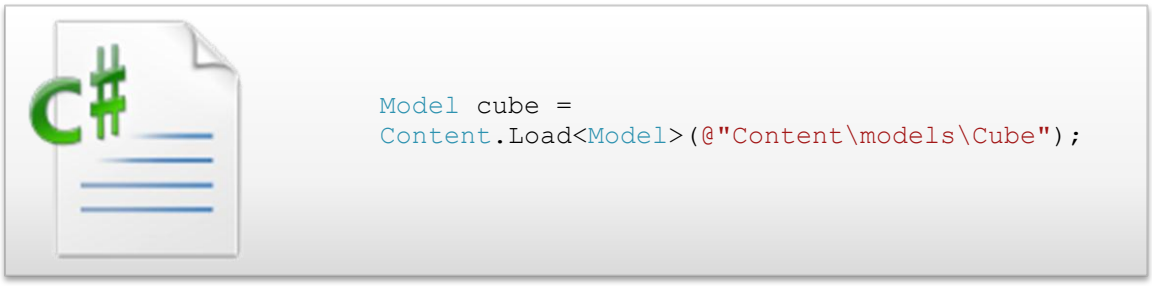
- Content.
- Documentation.
- GameLogic.
- GameScreens.
- Graphics.
- Helpers.
- Landscapes.
- Shaders.
- Sounds.
- Tracks.

4.4.1 Content

Esta carpeta no representa un namespace lógico dentro de los ensamblados del juego cuando es compilado, pero es necesaria su explicación y la inclusión dentro de la lista de namespaces.

Todos los objetos que no son código y que se requieren para la ejecución del juego se denominan recursos o *assets*. Los recursos en XNA son ficheros como archivos de música, sonidos FX, texturas o modelos tridimensionales, entre otras cosas que no se compilan y pasar a copiarse en la carpeta de la instalación del juego.

Cuando el programación realiza una carga de objetos por medio de la clase estática *Content* con el método genérico *Load<T>*, la carpeta por defecto donde busca el recurso es precisamente *Content.*



```
Model cube =
Content.Load<Model>(@"Content\models\Cube");
```

Código 30: Ejemplo de uso del método *Content.Load<T> ()*.

4.4.2 Documentation

La carpeta del proyecto *Documentation* contiene elementos de documentación relacionados tanto con el proyecto del juego como con el *starter kit* que genera el proyecto del juego.

4.4.3 GameLogic

Este namespace contiene todas las clases que controlan la física del juego, la manejabilidad de los vehículos en función de sus características de aceleración, tiempo de frenado y demás atributos, las distintas cámaras de visión y la captura de los diferentes dispositivos de entrada al juego como son el teclado, los mandos y la cámara TOF.

En el juego se ha añadido un nuevo controlador de los vehículos que viene dirigido por las acciones detectadas por la cámara. Estas acciones luego son transferidas a la lógica del juego y es éste quien modifica el juego.

En la imagen siguiente se muestra el conjunto de controladores que se pueden usar para realizar las funciones de volante en el juego. Este abanico de posibles controles representan un único objeto en la realidad: el volante.

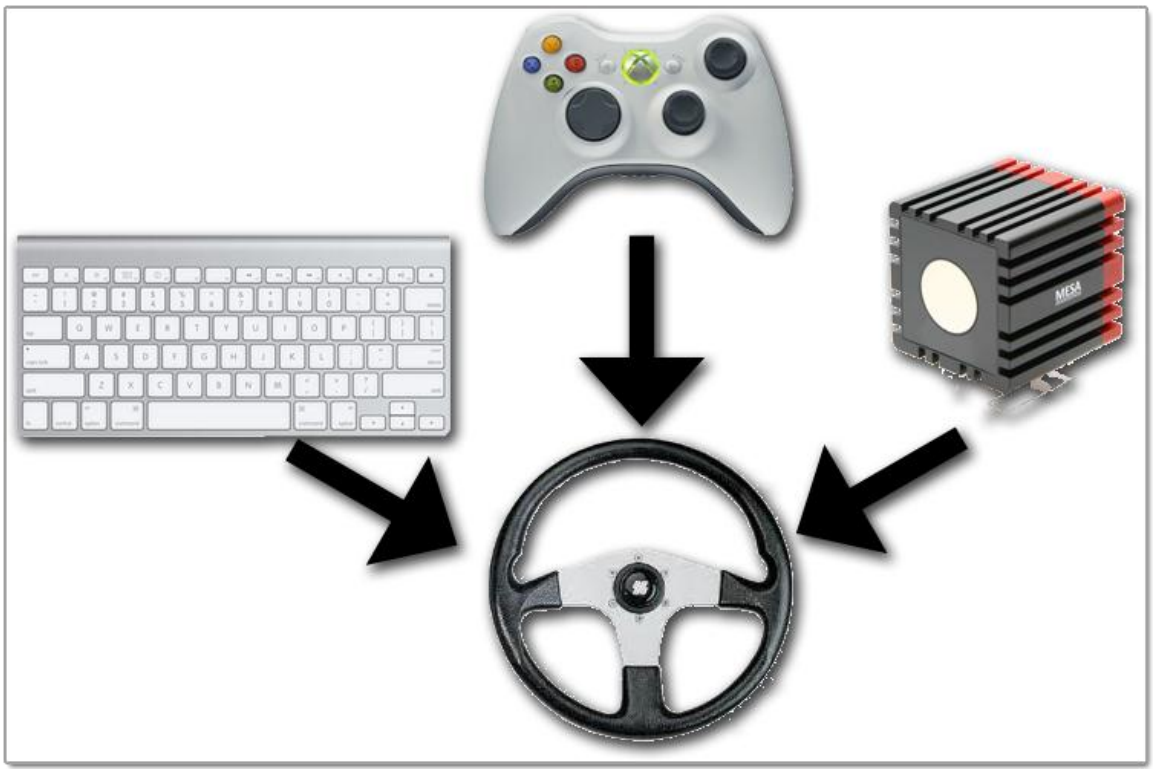


Figura 25: Controladores del juego Racing Game.

4.4.4 GameScreens

Este namespace contiene todas las clases que controlan la aparición de las pantallas de selección de pistas, selección de coches, opciones y demás pantallas de menú que cualquier juego presenta.

Entre estas pantallas se encuentra la propia pantalla de juego que presenta todos los elementos tridimensionales, efectos de luces, sonidos y demás características de un juego moderno.

A continuación se presentan las capturas de las pantallas que el jugador podrá visitar en el juego.

Este captura de pantalla muestras las opciones que el jugador puede realizar pulsando cada botón. Las acciones son:

- *Start race*: Pasa a la pantalla donde se tiene que seleccionar un vehículo para poder empezar a jugar.
- *Highscores*: Muestra las puntuaciones obtenidas en los tres niveles de juego que existen: *Beginner*, *Advanced* o *Expert*.
- *Options*: Muestra las posibles modificaciones que se pueden realizar en el juego sobre aspectos gráficos, de resolución o de sonido.
- *Help*: Muestra ayuda sobre los controles que manejan el vehículo.
- *Quit*: Abandona el juego.

En la captura aparecen dos textos mostrados por el juego. El primero que se observa es el número de fotogramas por segundos seguido de la resolución del juego. Algunas tarjetas gráficas no soportan ciertas características que el juego permite asignar al renderizado del videojuego. El segundo de ellos hace referencia a la detección o no de las manos sobre la cámara.

Figura 26: Pantalla inicial de Racing Game.

La siguiente captura muestra los resultados obtenidos por el jugador determinado. Aparece una lista de diez jugadores identificados con el nombre y el tiempo récord obtenido en la carrera. Estas puntuaciones aparecen clasificadas en los tres niveles de dificultad que permite el videojuego correr las carreras.

La pantalla presenta un botón abajo a la derecha para retroceder al menú principal de la aplicación mostrado en la Figura 26.

Figura 27: Highscores de Racing Game.

En la siguiente captura de pantalla se muestran las opciones principales del juego en cuanto a aspectos técnicos y gráficos así como características sonoras.

Los apartados que se presentan en la pantalla de opciones son:

- En primer lugar se presenta el nombre identificativo del jugador.
- Seguidamente se muestra el conjunto de resoluciones a las que se puede jugar. Estas resoluciones afectan en gran medida a la jugabilidad dependiendo de la tarjeta gráfica instalada en el ordenador.
- En el apartado gráfico se da a la opción al jugador de poner el juego a pantalla completa y de seleccionar los efectos de post-renderizado más comunes en los juegos de hoy en día. Se pueden activar las sombras, alta resolución de las texturas y efectos de post-renderizado que afecten a elementos ambientales del juego.
- Posteriormente se encuentra el apartado de sonido donde se puede modificar el volumen de los sonidos FX o ambientales producidos durante el juego.
- El próximo apartado determina el volumen de la música que se reproduce durante el juego.
- El último apartado que se presenta es la sensibilidad del controlador que determina el factor de escala o fuerza imprimida a la hora de producirse las acciones determinadas con el vehículo.

Figura 28: Options de Racing Game.

La imagen que aparece en la Figura 29 se muestra cuando el jugador en el menú principal decide comenzar una carrera. Antes de comenzar dicha carrera deberá pasar por dos filtros donde se seleccionará el vehículo con el cual quiere competir y la pista o nivel de dificultad donde quiere realizar la carrera.

En la pantalla de selección del vehículo se da la opción de poder optar a tres coches de competición que mostrarán modificaciones en los parámetros del motor. Los parámetros que un vehículo tiene son:

- Velocidad punta: es la velocidad máxima que alcanza el vehículo cuando se mantiene el acelerador.
- Aceleración: capacidad que tiene el motor de reaccionar y de llegar a la velocidad punta.
- Masa: peso que tiene el vehículo. Este atributo interfiere a la hora de realizar el frenado del vehículo debido a la inercia así como en la aceleración.
- Capacidad de frenado: capacidad que tiene el vehículo de frenar en función de la velocidad actual y de la masa.
- Fricción: capacidad que mide lo aerodinámico que es el vehículo y que interviene en el cálculo de la aceleración y velocidad punta.
- Motor: mide la calidad del motor y supone un factor de escala para los demás parámetros del vehículo.
- Color: determina el color de salida del vehículo.

La pantalla presenta abajo a la derecha dos botones que sirven para aceptar el vehículo seleccionado y configurado, y otro botón para volver al menú principal.

Figura 29: Choose your car de Racing Game.

Como ya se ha mencionado anteriormente, una vez el usuario haya seleccionado el vehículo con el cual quiere competir deberá pasar a la pantalla donde se seleccionará el nivel de dificultad que viene impuesto por la pista donde se rodará la carrera.

Existen tres niveles de dificultad que a su vez corresponden con tres ficheros de pistas o *Tracks*. Los niveles de dificultad son *Beginner*, *Advanced* o *Expert*.

De igual modo que en la pantalla anterior aparecen dos botones debajo a la derecha para aceptar la pista seleccionada o para volver a seleccionar un vehículo. Si se acepta la pista seleccionada se dará comienzo a la carrera.

Figura 30: Select Track de Racing Game.

Para concluir esta serie de capturas sobre el juego se presenta una donde se muestran el videojuego en plena acción con todos efectos de *shader* activados. Se puede observar como los reflejos del sol alteran la visión de la pantalla. Los efectos de desenfoque en función de la velocidad son notorios aumentando el grado de realismo. Por último, la carretera presenta un mapa de vectores normales que sirven como técnica mucho más compleja y efectista que el tradicional relieve o *bump mapping*.

En la pantalla se muestran los siguientes cuadros de mando:

- Vueltas: indica cuántas vueltas deben de correrse y cuál es el número actual vueltas dadas al circuito.
- Nivel: indica el nivel de la pista seleccionada.
- Tiempos: se muestran los cinco primeros tiempos que se han realizado en este circuito.
- Mejor Tiempo: muestra el mejor tiempo obtenido en el circuito.
- Tiempo Actual: muestra el tiempo obtenido por el jugador en dicha carrera.
- Velocímetro: este cuadro de mando muestra las revoluciones por minuto que lleva el motor para saber cuándo se debe pasar a la siguiente marcha, la marcha (*gear*) en la cual se encuentra el motor y la velocidad actual del vehículo.

Se debe recordar que el videojuego solo presenta el modo de conducción manual para el vehículo, lo que implica que el sistema detecta cuando se deben introducir las marchas.

Figura 31: Jugando a Racing Game.

4.4.5 Graphics

Este namespace presenta todo el conjunto de clases que renderizar elementos bidimensionales y tridimensionales. Las clases *Model*, *Texture*, *Font*, *LensFlare* son algunas que sirven para mostrar los objetos en el primer plano de la pantalla.

No todos los elementos renderizables se muestran en el mismo plano, sino que la imagen que el usuario percibe se compone de varias capas que se van agregando a una inicial.

4.4.6 Helpers

Los *Helpers* son clases estáticas que ayudan al programador a realizar tareas concretas. Dichas tareas no tienen por qué estar en varios objetos y se hace pesado implementar dichas clases bajo el patrón de diseño *Singleton*.

Las clases estáticas, al permanecer en memoria desde que da comienzo la aplicación hasta que se abandona, suelen ejecutarse mucho más rápido, y así no se obliga a la máquina virtual a estar constantemente pendiente de dichos punteros a las estructuras estáticas para posteriormente ser liberados.

4.4.7 Landscapes

Este namespace se encarga de gestionar todos los objetos tridimensionales y su meta-información relacionada con los buffers que almacenan los vértices de los elementos que conforman el entorno donde se emplaza el circuito.

En el juego sólo se encuentra un *landscape* o entorno que simula las condiciones de un desierto. Todos los objetos que existen en él como cactus, arena, dunas, palmeras forman el *atrezzo* de dicho entorno.

Los programadores pueden modificar dichos elementos para adaptarlos a otras condiciones o ecosistemas.

4.4.8 Shaders

Un *shader* es un código para el sombreado que se ejecuta directamente en la tarjeta gráfica y que permite realizar efectos muy potentes y realistas sobre los elementos tridimensionales que existen en la escena.

Existen diferentes tecnologías que implementan *shaders* como HLSL de Microsoft y Nvidia, GLSL para OpenGL o CG también de Microsoft y Nvidia para tarjetas concretas basadas en perfiles.

Estas tres tecnologías permiten realizar cálculos complejos y la interacción sobre los modelos tridimensionales de la escena, así como los elementos bidimensionales (texturas).

En este namespace, por tanto, se almacenan el conjunto de *shaders* que se aplican los objetos, materiales o texturas del videojuego. La tecnología implementada para lograr la aplicación de los efectos es HLSL basada en *PixelShader*.

4.4.9 Sounds

Este namespace se encarga de la parte sonora del videojuego. Como en todo juego que se precie, el conjunto de sonidos se divide en dos:

- Sonidos FX: estos sonidos son efectos que se producen cuando se producen acciones determinadas en el juego. Cuando el vehículo acelera, cuando frena o cuando de marcha, por ejemplo, se reproducen sonidos que ayudan a aumentar el grado de realismo del juego.
- Música: estos sonidos permanecen constantes a lo largo del desarrollo del juego y pueden repetirse una y otra vez, sin que el usuario realice ninguna acción.

El paquete de sonidos del videojuego se reproduce en una arquitectura propia que se denomina XACT que está presente desde Windows Vista. Esta colección de librerías viene a sustituir a la API obsoleta *Direct Sound*.

4.4.10 Tracks

Este namespace se encarga de la gestión de todos los elementos que forman parte de un circuito. No se debe confundir el circuito con el ecosistema que forma parte de la misma carrera.

En el circuito se encuentran los edificios alrededor del circuito, la carretera con sus curvas y rectas, los puntos de control (*checkpoint*) y un largo etcétera de objetos que el programador puede quitar o agregar a su antojo.

4.5 Volante Virtual

Antes de describir la clase Volante que se ha utilizado en el desarrollo del juego es conveniente aclarar las posiciones y acciones que deberá realizar el usuario del volante virtual y que se transferirán al videojuego.

El volante detecta la situación donde se encuentran los puños del jugador en la pantalla. Cuando se tienen los puños detectados, se calcula el centro de masas para cada uno de ellos obteniéndose una única coordenada por cada mano o puño. Dependiendo de dónde se sitúen las coordenadas de cada uno de los centroides de los puños, se ejecutará a una acción u otra.

Las acciones que se pueden realizar son:

- Acelerar el vehículo.
- Girar a la derecha el vehículo.
- Girar a la izquierda el vehículo.
- Frenar el vehículo.

En la siguiente figura se muestra una captura de la interfaz gráfica presentanda en el Capítulo 3 donde se observan los centroides capturados y la pose del usuario tomando un volante inexistente.

Los parámetros impuestos para capturar la imagen han sido un *Threshold* de 528 y un tiempo de intregación de 3 milisegundos.

Como se observa existen tres colores predominantes en la imagen central de la aplicación. Estos colores representan las distancias donde se encuentran cada uno de los pixeles capturados.

El color verde implica que los pixeles han superado el umbral o *Threshold* establecido y tras aplicar dicho umbral quedan fuera del plano. Esto constituye una forma de eliminar pixeles o regiones innecesarias para el tratamiento de la imagen.

El color blanco indica que las zonas o regiones captadas con ese color están dentro del plano pero muy lejos de la cámara. Esta disección del plano en colores ayuda a localizar las distancias, y consecuentemente a dispensar o clasificar unas regiones de otras. En este caso el color blanco también resulta innecesario y no se utiliza para realizar los cálculos sobre la imagen.

El color gris, que comienza a partir de color umbral, indica que todos los canales de color (*RGB, Red Green Blue* – Rojo Verde Azul) de cada píxel poseen el mismo valor. Con esta información y a partir de valor umbral interno se calculan los centroides para cada uno de los puños.

Los centroides en rojo son un mapa adicional al canal capturado por la cámara. Esto representa un mapa de control o ayuda y no interfiere en los cálculos del volante virtual.

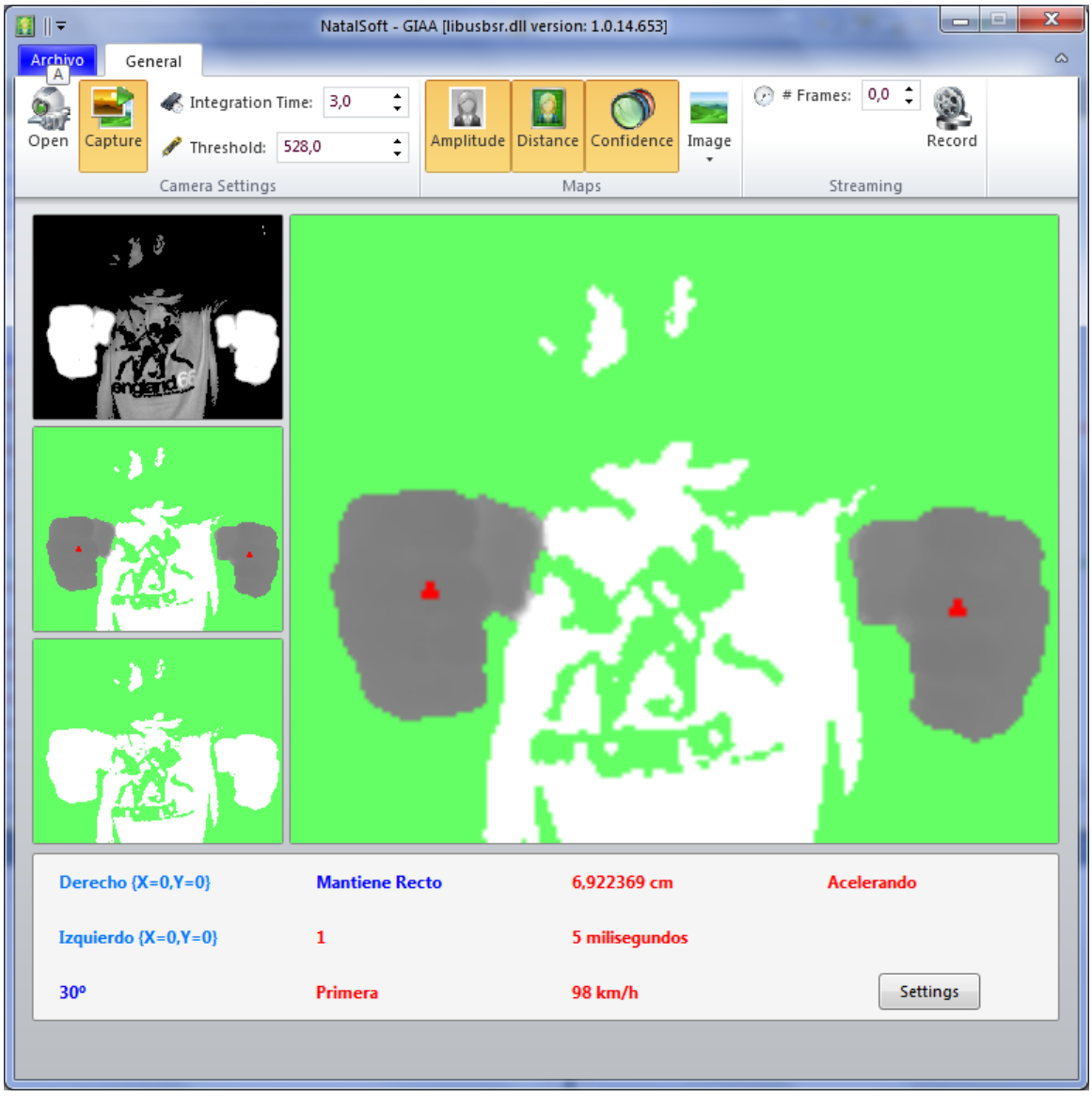


Figura 32: Imagen de los centroides capturados.

4.5.1 Segmentación de la visión

Para poder catalogar las acciones del usuario se ha segmentado la pantalla en zonas activas que representan acciones que el jugador trasladará al vehículo. Las zonas que existen son:

- Zona de aceleración.
- Zona de frenado.

La zona de frenado corresponde a la zona roja de la pantalla capturada, mientras que la zona azul representa la zona de aceleración. Si los dos centroides se encuentran en la

zona roja, el vehículo frenará, mientras que si los dos centroides se encuentran en la zona de aceleración, el vehículo aumentará la velocidad.

Si los centroides no están en la misma zona, el vehículo se mantendrá inmóvil, y si éste llevaba velocidad alguna, sólo se moverá por inercia. Si el jugador quiere parar el motor (ni frenar ni acelerar) deberá mantener los centroides en zonas distintas.

Dado que la resolución de la cámara es muy pequeña (176 pixeles de ancho por 144 pixeles de alto) es preferible realizar dos zonas (frenado y aceleración) que no tres (frenado, aceleración, no-acción).

En la siguiente figura se muestra como está segmentada la imagen que devuelve la cámara para procesar y aplicar las acciones determinadas en función de la posición de los centroides.

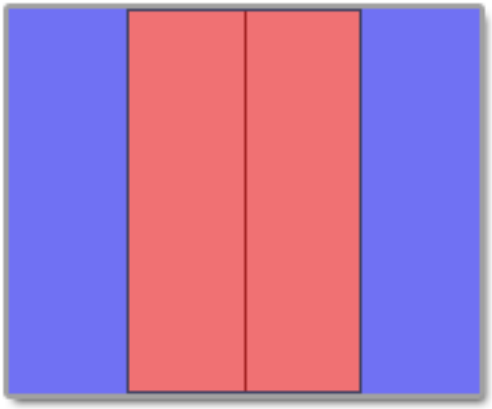


Figura 33: Segmentación del volante virtual.

Seguidamente se muestran tres figuras donde se detalla el comportamiento que ofrece el volante en función de la posición de los centroides.

Existe un rectángulo formado por la superficie que ocupan las dos diagonales. Este rectángulo morado establece un umbral para el cual el volante se encuentra en movimiento recto (sin giros) aunque la pendiente de la recta tangente entre los dos centroides no sea exactamente 180.

Este área morada ofrece un grado de libertad a la hora de controlar el volante para mantenerlo recto. Las dos diagonales que están inscritas en el rectángulo representan las coordenadas máximas en las que pueden estar los centroides, respectivamente a cada diagonal.

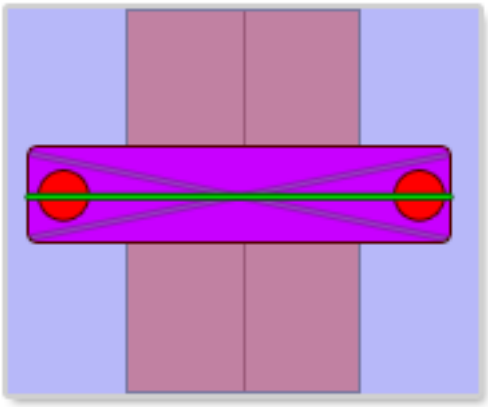


Figura 34: Umbralizador de giros.

La siguiente acción a explicar contempla el giro del volante virtual. Este giro se realiza cuando la pendiente entre los dos centroides abandona el rectángulo morado presentando en la Figura 34.

Si la pendiente es negativa, el giro se realiza a la derecha, para lo cual, el jugador deberá subir el puño izquierdo y bajar el puño derecho. Tal y como está programado el umbral de giros, siempre y cuando, la recta formada por los dos centroides abandone la región morada, se estará en función de un giro, a izquierdas o derechas, relativo al signo de la pendiente.

En cuanto a la intensidad, cabe esperar que a menor pendiente, mayor intensidad en el giro. Esta opción se ha eliminado de la programación en el videojuego, ya que éste está enfocado a entradas de teclado o mandos de la XBOX. Los botones tienen dos estados: pulsado o no pulsado en un momento determinado de la acción. Al presentar dos estados, la escala posible de giros o acciones en general se ve limitada. La acción de giro a la izquierda es análoga a la presentada en la Figura 35 donde el centroide izquierdo esta abajo y el derecho está arriba.

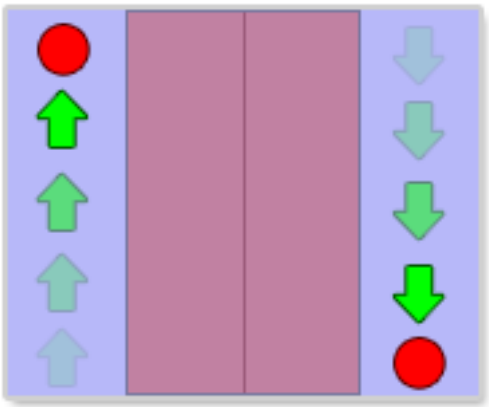


Figura 35: Girar el volante a la derecha.

La última acción importante que presenta el volante es la desaceleración o frenado del vehículo. La acción de frenado se produce cuando los dos centroides entran en la zona de frenado (en rojo). Cabe esperar que la acción de frenado también tenga grados de intensidad, al igual que lo esperado en los giros, pero es la misma excusa por la que se obvia dicha característica.

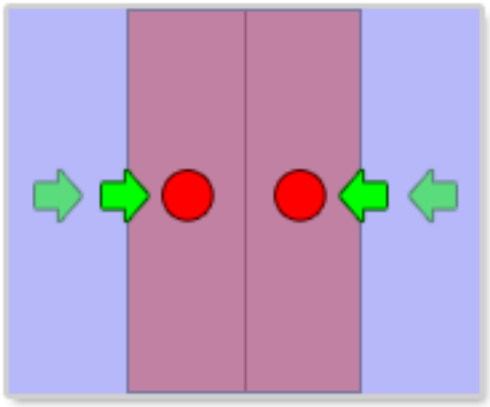


Figura 36: Frenar el vehículo.

Con las acciones presentadas, se pasa a describir la clase Volante que encapsula el manejo de la cámara y clases anteriormente vistas. La clase Volante encapsula a la *Clase SRCamera* explicada en el Capítulo 2 y ofrece una interfaz sencilla sobre la posición de los centroides y su aparición o no en la pantalla.

Este conjunto de acciones puede ser combinable de muchas varias formas. Como por ejemplo, el usuario puede estar girando a la derecha mientras que está frenando o puede acelerar mientras gira en un sentido. Para ello en la clase que abstrae el comportamiento del volante se han dividido las acciones de giro y las acciones de aceleración o frenado en dos enumeraciones distintas:

- Enumeración *ActionCar*.
- Enumeración *DirectionCar*.

4.5.2 *ActionCar*

La enumeración *ActionCar* representa el conjunto de acciones que tiene que ver con la aceleración o el frenado y con la detección de los centroides en las capturas desde la cámara.

La colección de acciones que presenta la enumeración es la siguiente:

- Up: indica que los dos centroides están en la zona de aceleración y por tanto, el vehículo tenderá a aumentar su velocidad.
- Down: indica que los dos centroides están en la zona de frenado y por tanto, el vehículo tenderá a disminuir su velocidad.
- StandBy: indica que uno de los centroides se encuentra en la zona de frenado y el otro se encuentra en la zona de aceleración. Es indistinto para establecer esta acción en el volante cuál de los dos centroides se encuentra en la zona de aceleración y cual en la zona de frenado.

-  Not_Left: indica que el centroide izquierdo no se encuentra en el plano. Esto se traduce en un cambio de marcha hacia arriba. Esta funcionalidad se usa en el juego debido a que los cambios de marcha son automáticos.
-  Not_Right: indica que el centroide izquierdo no se encuentra en el plano. Esto se traduce en un cambio de marcha hacia arriba. Esta funcionalidad se usa en el juego debido a que los cambios de marcha son automáticos.
-  Not_Detecting: indica que los dos centroides no se encuentran en plano de captación de la cámara. Si esto ocurre, no se muestran en la pantalla y el usuario rápidamente podrá volver a situar los puños en la distancia y ángulo correcto.

4.5.3  *DirectionCar*

La enumeración *DirectionCar* representa el conjunto de acciones relacionadas con la dirección del giro o del volante.

La colección de acciones que presenta la enumeración es la siguiente:

-  None: indica que la pendiente de los centroides no ha abandonado la región morada utilizada como umbral para mantener el volante recto.
-  Left: indica que la pendiente de los centroides es negativa y que el volante gira hacia la izquierda.
-  Right: indica que la pendiente de los centroides es positiva y que el volante gira hacia la derecha.

4.5.4  La clase Volante: Consideraciones, Calibrado y Constantes

La primera consideración a tener en cuenta es que la clase Volante implementa el patrón de diseño *Singleton*. Es especialmente útil ya que mantiene la estructura de la clase de forma estática acelerando los cálculos y liberando a la máquina virtual de .NET de realizar ciertas tareas mientras el videojuego está en ejecución.

De esto forma se pueden tener varias clases de volante que representen varias cámaras a la vez de forma estática.

En XNA esta función está integrada en una estructura de datos estática que encapsula el teclado o los mandos que puedan estar conectados a la XBOX.

Otra consideración importante es tener en cuenta que las variables del juego han tenido que ser recalibradas por la incorporación del volante virtual. El hecho imperante se

deduce que en el teclado o en los mandos las acciones se pulsan, mientras que en el volante virtual se pueden mantener y se pueden hacer constantes con una intensidad determinada.

Para entender mejor la calibración de los parámetros del juego en función del volante se describen las constantes o factores que afectan de manera inmediata dichas variables de control.

La colección de constantes públicas que afectan al calibrado del videojuego es la siguiente:

- REDUCE_GIRE: representa la corrección en el ángulo de giro del vehículo que se debe realizar para el giro sea suavizado.

```
public const uint REDUCE_GIRE = 5;

[…]   // Código de la clase CarPhysics.

virtualRotationAmount -= interpolatedRotationChange
/ Volante.REDUCE_GIRE;
if (isCarOnGround)
      carDir = Vector3.TransformNormal(carDir,
            Matrix.CreateFromAxisAngle(carUp,
            interpolatedRotationChange /
            Volante.REDUCE_GIRE));

[…]   // Continúa el código de la clase.
```

Código 31: Incorporación del volante virtual a los giros del vehículo.

En este extracto de código del juego se observa como las variables *virtualRotationAmount* y *interpolatedRotationChange* son escaladas por la constante *REDUCE_GIRE* que este caso tiene un valor de cinco.

- REDUCE_SPEED: representa la corrección en la velocidad del vehículo que se debe realizar para las curvas se puedan tomar de forma más suavizada.

```
public const uint REDUCE_SPEED = 5;

[…]   // Código de la clase CarPhysics.

speed += speedChangeVector.Length() *
speedApplyFactor / Volante.REDUCE_SPEED;

[…]   // Continúa el código de la clase.
```

Código 32: Incorporación del volante virtual a la velocidad del vehículo.

En este extracto de código del juego se observa como la variable *speed* es escalada por la constante *REDUCE_SPEED* que en este caso tiene un valor de cinco.

- HALF_SCREEN: representa la mitad de la pantalla en las dimensiones de la cámara TOF que captura la escena. Este valor es siempre 88 que corresponde a la mitad del número de pixeles en la coordenada horizontal de la imagen de la cámara.

```
public const uint HALF_SCREEN = 88;

[…]   // Código de la clase UIRenderer.

TextureFont.WriteTextCentered(input3D.RightPoint.X +
      BaseGame.Width / 2 + (int)Input3D.HALF_SCREEN,
      BaseGame.YToRes(26), actionString);

[…]   // Continúa el código de la clase.
```

Código 33: Incorporación del volante virtual al pintado de los centroides.

4.5.5 La clase Volante: propiedades

Las propiedades que ofrece la cámara están compuestas de variables de control sobre los centroides, de la posición de los mismos o bien de las enumeraciones *ActionCar* y *DirectionCar* anteriormente explicadas.

El conjunto de propiedades que proporciona el volante es la siguiente:

- Action: determina un valor representado por la enumeración *ActionCar* en función de la posición de los centroides.
- BrakingIntensity: determina el grado de intensidad de la frenada en función del inverso de la distancia de los centroides si el valor de la propiedad *Action* es *ActionCar.Down*.

```
public float BrakingIntensity
{
      get {
            if (Action == ActionCar.Down)
                  return 1 / Intensity;
            else
                  return 0;
      }
}
```

Código 34: Creación de la intensidad de frenado.

- Camera: determina una instancia estática de la clase *SRCamera*. Esta variable *SRCamera* es la que se encarga de procesar y capturar los fotogramas de la escena del usuario.
- Direction: determina un valor representado por la enumeración *DirectionCar* en función de la posición de los centroides.
- Handle: determina un identificador donde la cámara puede enlazar como objeto paterno cuando tiene que dibujar los cuadros de diálogo internos que se muestran en la Figura 11 y en la Figura 12.
- Intensity: determina el grado de intensidad de la frenada en función de la distancia de los centroides si el valor de la propiedad *Action* es *ActionCar.Up*.

```
public float Intensity
{
      get {
            if (Action == ActionCar.Up)
                  return (right.X - left.X) /
                        cam.Width;
            else
                  return 0;
      }
}
```

Código 35: Creación de la intensidad de las acciones en el volante virtual.

- IsDown: determina si acción del vehículo que viene determinada por la enumeración *ActionCar* es igual a *ActionCar.Down*. Esta propiedad, al igual que otras posteriores, ayudan a la mantener la coherencia con los distintos tipos de interfaces de control del vehículo dentro de la programación del videojuego. Un ejemplo de ello es:

```
Input3D input3D = Input3D.GetInstance(new IntPtr());

[…]   // Código de la clase CarPhysics.

if (Input.KeyboardDownPressed ||
      Input.Keyboard.IsKeyDown(Keys.S) ||
      Input.Keyboard.IsKeyDown(Keys.O) ||
      Input.MouseRightButtonPressed ||
      input3D.IsDown)
            newAccelerationForce -=
                  maxAccelerationPerSec;

[…]   // Continúa el código de la clase.
```

Código 36: Incorporación del volante virtual al cambio de la velocidad.

En este ejemplo de código se muestra como se captura la señal de las interfaces o dispositivos de entrada al videojuego (teclado, mandos o cámara TOF). En este caso se produce una frenada que decrementa la variable *newAccelerationForce*.

-  IsLeft: determina si acción del vehículo que viene determinada por la enumeración *DirectionCar* es igual a *DirectionCar.Left*.
-  IsNotDetecting: determina si acción del vehículo que viene determinada por la enumeración *ActionCar* es igual a *ActionCar.NotDetecting*.
-  IsNotLeft: determina si acción del vehículo que viene determinada por la enumeración *ActionCar* es igual a *ActionCar.Not_Left*. Este valor no debe interpretarse como que la acción del vehículo es no girar a la izquierda, sino que determina que el centroide o puño izquierdo no se ha capturado porque el usuario ha ejecutado dicha acción.
-  IsNotRight: determina si acción del vehículo que viene determinada por la enumeración *ActionCar* es igual a *ActionCar.Not_Right*. Este valor no debe interpretarse como que la acción del vehículo es no girar a la derecha, sino que determina que el centroide o puño derecho no se ha capturado porque el usuario ha ejecutado dicha acción.
-  IsRight: determina si acción del vehículo que viene determinada por la enumeración *DirectionCar* es igual a *DirectionCar.Right*.
-  IsUp: determina si acción del vehículo que viene determinada por la enumeración *ActionCar* es igual a *ActionCar.Up*.
-  LeftPoint: esta propiedad viene determina por una estructura de datos del namespace *System.Drawing.Point* y ofrece las coordenadas donde se encuentra situado el centroide izquierdo.
-  RightPoint: esta propiedad viene determina por una estructura de datos del namespace *System.Drawing.Point* y ofrece las coordenadas donde se encuentra situado el centroide derecho.

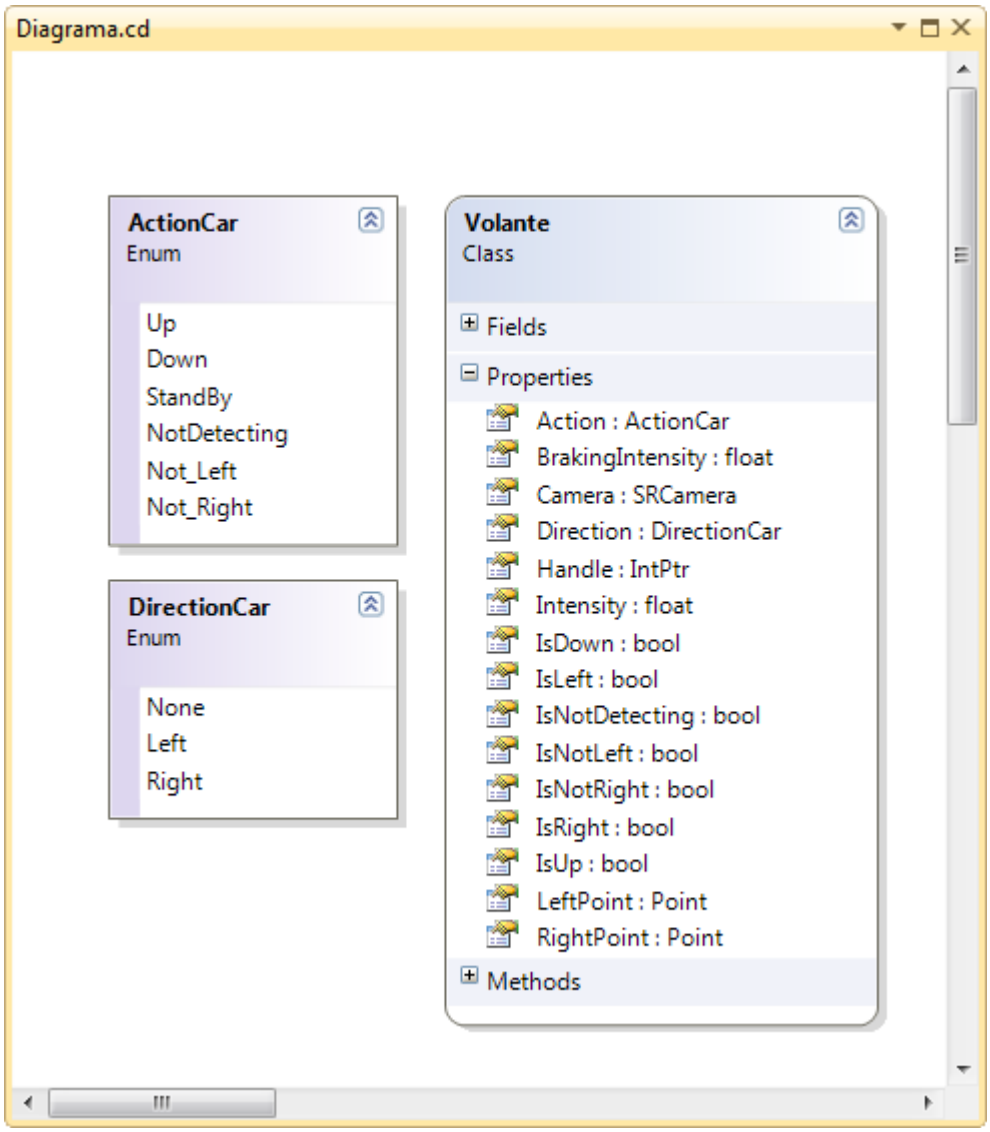


Figura 37: Propiedades del volante virtual.

4.5.6 La clase Volante: métodos

Por último, se presentan todos los métodos que hacen posible el comportamiento de la clase Volante. De todos los métodos existentes sólo tres son públicos para encapsular la funcionalidad de la clase.

La colección de métodos que ofrece la cámara es la siguiente:

- AcquireAndDisplay (): este método se encarga de conseguir el conjunto de píxeles en sus coordenadas tridimensionales por medio de la llamada *SRCamera.GetCartisian ()* previa ejecución de la función *SRCamera.Acquire ()*. Una vez conseguidas las coordenadas cartesianas, se debe crear a partir de ellas, el canal de las distancias, que es el único funcional para llevar a la práctica la utilidad del volante.

En el caso en que se pueda realizar la ejecución de alguno de los movimientos deducidos y almacenados por medio de las dos enumeraciones: *ActionCar* y *DirectionCar*, se pasará a su ejecución. Siendo éste el método más importante de la clase se muestra la parte fundamental del código.

```
[…]   // Código de la clase Volante.

if (camera.Acquire() < 0)
      return;

camera.GetCartesian(out xvalues, out yvalues, out
zvalues);
distance = camera.GetImage(ImgType.IT_DISTANCE);

bm = DrawArrayToBitmap(distance, camera.Width,
maxDistance);
if (miliseconds >= TIME_ACTION)
{
      DetectMovement();
      DetectDirection();
}
miliseconds++;

[…]   // Continúa el código de la clase.
```

Código 37: Código esencial de la clase Volante.

- DetectDirection (): este método se encarga de deducir el valor de la dirección por medio de la enumeración *DirectionCar*. Este método se ejecuta a intervalos de tiempo que el programador establece en el valor constante *Volante.TIME_ACTION*.
- DetectMovement (): este método se encarga de deducir el valor del movimiento por medio de la enumeración *ActionCar*. Este método se ejecuta a intervalos de tiempo que el programador establece en el valor constante *Volante.TIME_ACTION*
- DrawArrayToBitmap (): este método se encarga de pintar los pixeles obtenidos por la llamada *SRCamera.GetImage (ImgType.IT_DISTANCE)*. Para ello, recibe como parámetros de entrada una imagen de clase *Bitmap* y un *array* que determina el conjunto de pixeles a pintar en la imagen.
- GetInstance (): este método devuelve la única instancia que existe en el código para la cámara TOF en base a una dirección IP. Para lograr el objetivo de mantener una única se ha implementado la clase basándose en el patrón de diseño *Singleton*.
- InitMovements (): este método se encarga de establecer el valor de las dos enumeraciones a valores por defecto.

- MobilMean (): este método se encarga de calcular la media móvil de las distancias entre los centroides para averiguar su intención de movimiento. El cálculo de la media móvil se realiza cada milisegundo con un conjunto de diez valores.
- DrawPoint (): este método se encarga de pintar los puntos de los centroides de una imagen determinada. Para ello el método recibe como parámetros la imagen donde se quieren pintar el centroide y la coordenada cartesiana del centroide.
- Volante (): este método privado constituye el constructor de la clase. Para implementar correctamente el patrón de diseño *Singleton* se debe establecer la visibilidad del constructor a privado.

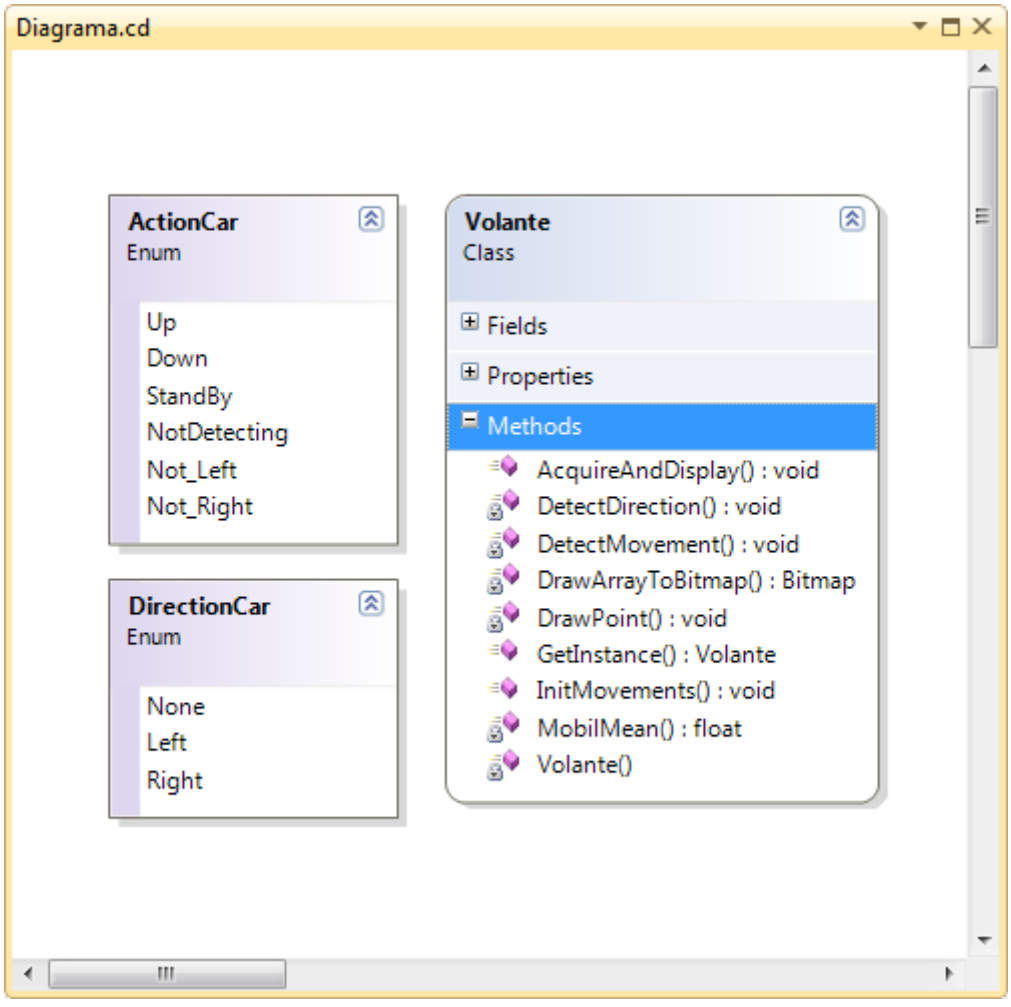


Figura 38: Métodos del volante virtual.

Capítulo 5

Conclusiones

5.1 Claves

La claves para este capítulo son:

- Análisis de los objetivos logrados.
- Creación de modelos 3D.
- Líneas futuras del proyecto.

5.2 Introducción

En esta sección del documento se presentan las conclusiones alcanzadas tras la finalización del proyecto en base a cada uno de los objetivos establecidos inicialmente.

A continuación se recuerdan los objetivos planificados para el proyecto:

- Investigar la tecnología TOF y sus aplicaciones: esta parte del proyecto ha sido la más enriquecedora ya que ha permitido descubrir nuevas áreas de aplicación de tecnologías en auge que antes eran desconocidas.

- Investigar la cámara TOF SR4000: una vez conocida la tecnología se debe concretar en la implementación específica que ha realizado Mesa Imaging en su producto SR4000. Aunque los cálculos teóricos sobre las ecuaciones matemáticas imperantes en el dominio de aplicación se han respetado, el flujo de procesado por medio de canales y filtros, y el conocimiento del funcionamiento interno de la cámara ha centrado la atención de los estudios también en áreas irrelevantes.

- Diseñar y documentar una API de comunicaciones con la cámara TOF: este objetivo ha sido el más complejo de realizar y comprender ya que el paso de las explicaciones teóricas del funcionamiento de la cámara a la aplicación directa de las mismas, no ha presentado un camino natural o lógico de realizar.

- Implementar una aplicación TOF que haga uso de la API: una vez obtenidos los objetivos anteriores, el desarrollo de una aplicación que saque provecho de los análisis ha sido una tarea que ha puesto de manifiesto la consecución obvia de la planificación del proyecto.

- Implementar una aplicación orientada al ocio con la cámara SR4000: este objetivo ha resultado el menos costoso de realizar, y sin embargo, el más aparentemente productivo de todos, ya que ha permitido observar el funcionamiento de la cámara en aplicaciones reales con un impacto directo y visible.

5.3 Creación de modelos 3D

Esta sección representa uno de los sub-objetivos parciales del proyecto y que consiste en crear una malla tridimensional a partir de los pixeles que ofrece la cámara. Se recuerda que la función *SRCamera.GetCartesian ()* obtiene la información tridimensional para cada uno de los pixeles de la escena.

El procedimiento es bien sencillo de realizar, aunque su implementación pueda ser mucho más costosa. Los pasos propuestos para la ejecución del algoritmo son los siguientes:

- Obtener una imagen de la escena sin el usuario con el mayor tiempo de integración posible. Al aumentar mucho el tiempo de integración se obtienen medidas de los pixeles con mucha certidumbre.
- Seguidamente se obtiene la imagen de escena, ahora con el usuario en ella, y con un tiempo de integración acorde a las necesidades impuestas por los criterios de velocidad o rendimiento.
- Con las dos imágenes, una sólo de la escena ,y otra con la escena y usuario, se aplica un umbralizador entre las dos imágenes donde los pixiles coincidentes son eliminados. Se debe tener en cuenta que la imagen con el usuario presenta incertidumbre en sus pixeles debido al uso de un tiempo de integración menor, lo cual exige un tratamiento de búsqueda de coordenadas similares en el espacio tridimensional.
- En este paso se tiene un modelo de pixeles que sólo contemplan al usuario. Ahora es el momento de crear la malla tridimiensional, lo cual requiere del cálculo de normales para observar dicha malla y la ordenación y nombrado de los índices de los triángulos que conforman dicha malla. Los índices indican el orden en el cual se pintan los triángulos y qué vértices contiene cada triángulo. Esta tarea no es trivial. Se recuerda que en los modelos de representación tridimensional, el bloque fundamental de construcción es el triángulo.
- Con la malla generada ahora queda introducir los huesos al modelo. Estos huesos o más bien, las articulaciones de los mismos, permitirán saber los ángulos de giro y la posición exacta de cada parte del cuerpo. Con esta información ya se podría detectar los movimientos realizados por los humanos y generar entornos virtuales más complejos y dinámicos.

Se muestra una captura de un modelo tridimensional completo de la anatomía humana. Esta imagen debe representar el modelo final obtenido por la consecución de dicho objetivo, pero hay que tener en cuenta un factor importante: la cámara solo muestra y captura la malla humana visible a ella, y no todo el cuerpo.

Esto quiere decir que si el usuario está de frente a la cámara, la espalda del mismo queda eclipsada por la parte frontal. El producto Kinect presentado sólo se conforma con esta malla y no requiere del cuerpo humano completo.

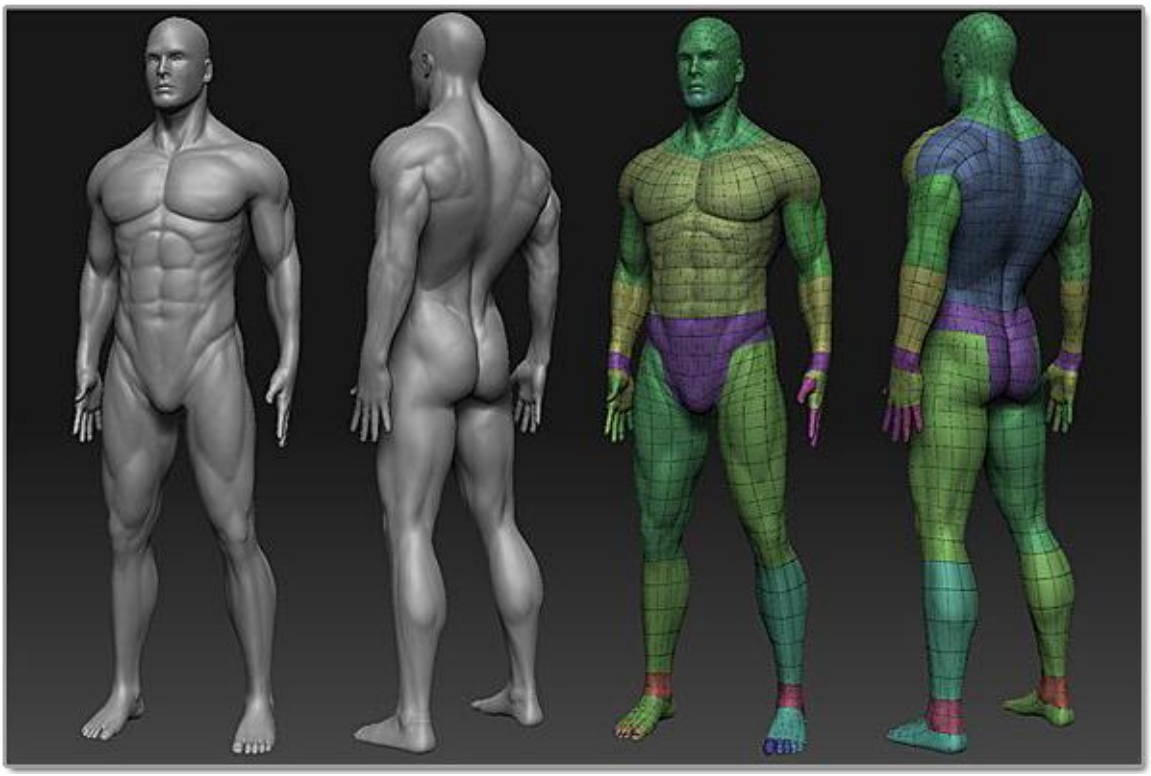


Figura 39: Malla de un modelo humano.

Para crear una malla tridimensional humana se requiere de al menos dos cámaras TOF conectadas y enfrentadas donde la escena queda en medio de ambos planos. Cada cámara creará su versión del humano y deberá existir un paso final para unir las dos mitades en un único modelo tridimensional.

5.4 Líneas futuras

La Biometría es la ciencia que estudia la identificación y verificación de rasgos o características físicas e intransferibles de las personas, como por ejemplo la huella digital. Las cámaras TOF pueden ayudar en esta área capturando la estructura facial en tres dimensiones de los individuos, para posteriormente ser verificados en un sistema.

La extracción de la información a identificar se realiza de forma sencilla y no muy intrusiva como pueda ser la lectura del iris. Un avance importante y definitorio en la imagen capturada es que los pixeles no representan colores sino profundidades o distancias. Hasta el momento, las imágenes tenían grados de variabilidad debido a la cantidad de luz o sombra que existía en la escena. La profundidad no tiene en cuenta estos factores aunque sí otros. Las zonas con reflexiones altas pueden distorsionar las trayectorias de los fotones que deben ser capturados para crear la malla tridimensional. Si estos fotones son reflejados con ángulos muy altos, los pixeles no almacenarán la información real de la escena. La ventaja es que es más fácil crear una escena con baja reflexión que eliminar las zonas de luz y sombra en una imagen tradicional.

A continuación se muestra una comparativa entre la imagen de profundidad y una termografía de la cara que también presenta sus singularidades debido a la tempetura interna al individuo (ha realizado ejercicio, acaba de comer, etc.) o externa (verano, invierno, calefacción, aire acondicionado, hay focos de luz en la escena, etc.).

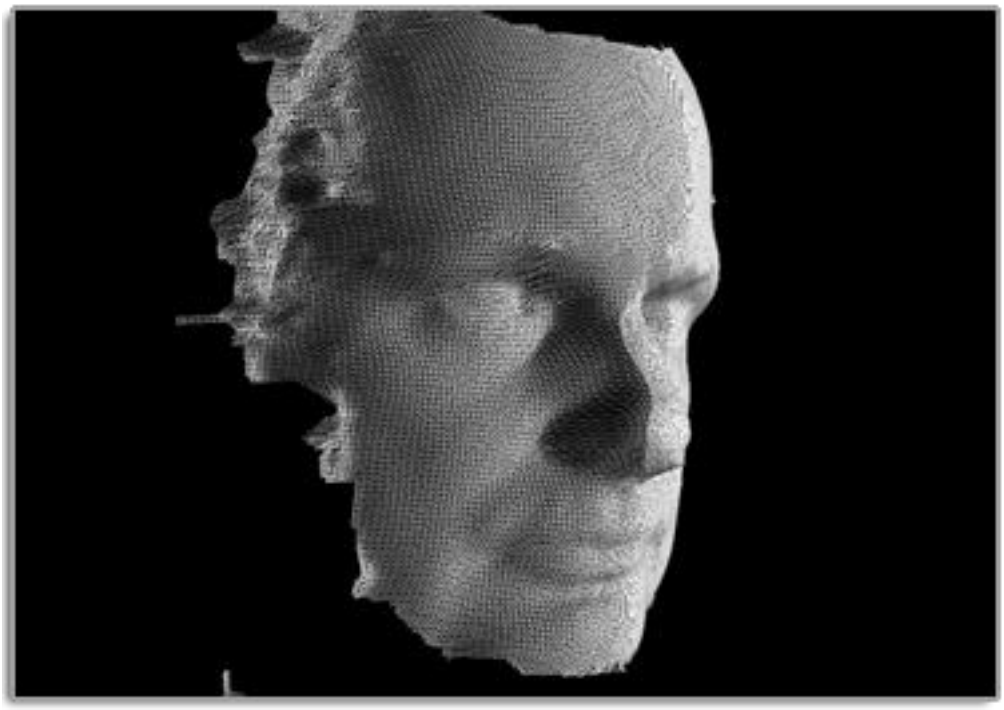


Figura 40: Estructura facial en tres dimensiones.

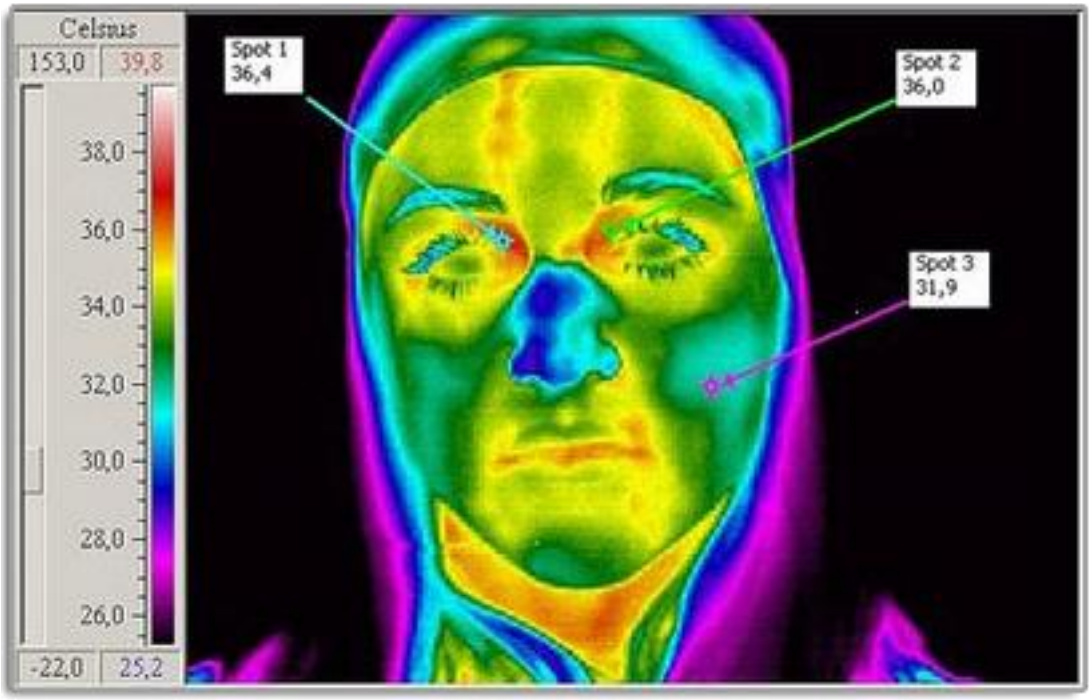


Figura 41: Termografía de una cara humana.

Una de las ciencias más abiertas a la incorporación o sinergia con otras ciencias es la Medicina. Esta ciencia se encarga de mantener o curar la salud de los seres vivos, y más concretamente de los seres humanos.

Cada vez más se realizan logros tangibles con la fusión de las nuevas tecnologías y la medicina. Un ejemplo de ello es un escáner creado por Philips (denominado *Brilliance iCT Scanner*) que incorpora la tecnología de rayos X para crear un modelo 4D en tiempo real del cuerpo humano. El modelo tetradimensional (*tetramomentum* donde se definen las tres coordenadas espaciales de los pixiles y la capa de profundidad o tejido) presente en el cuerpo donde aparecen las venas, los huesos, los músculos y todos los demás órganos de la anatomía humana.

Las siguientes muestran los resultados de dicho escáner:

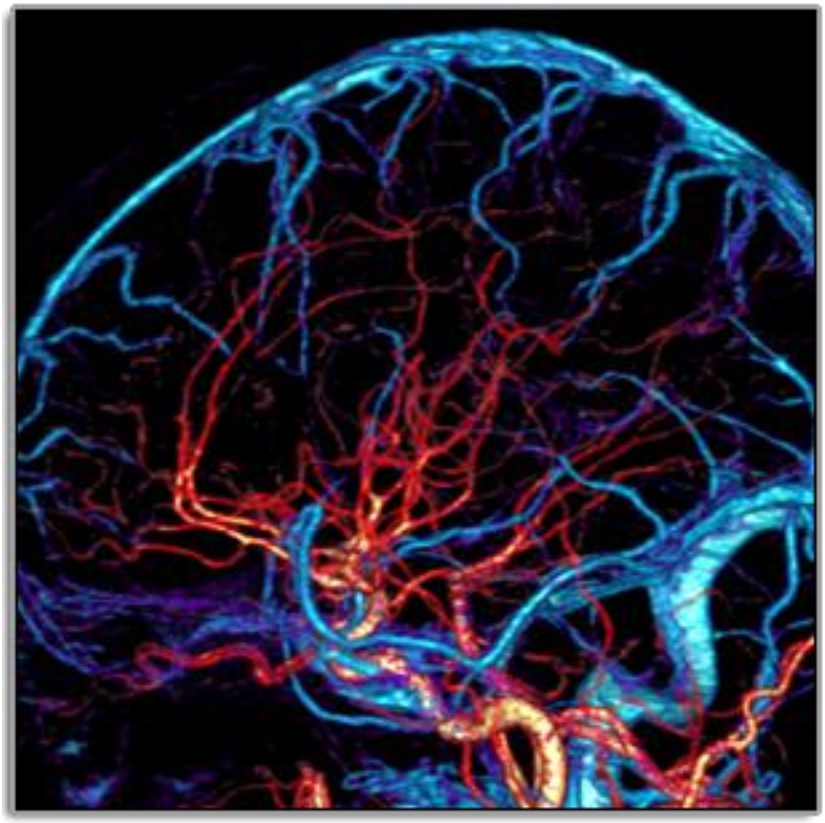


Figura 42: Escaner 4D de los vasos que irrigan el cerebro.

Figura 43: Escáner 4D del exterior del corazón.

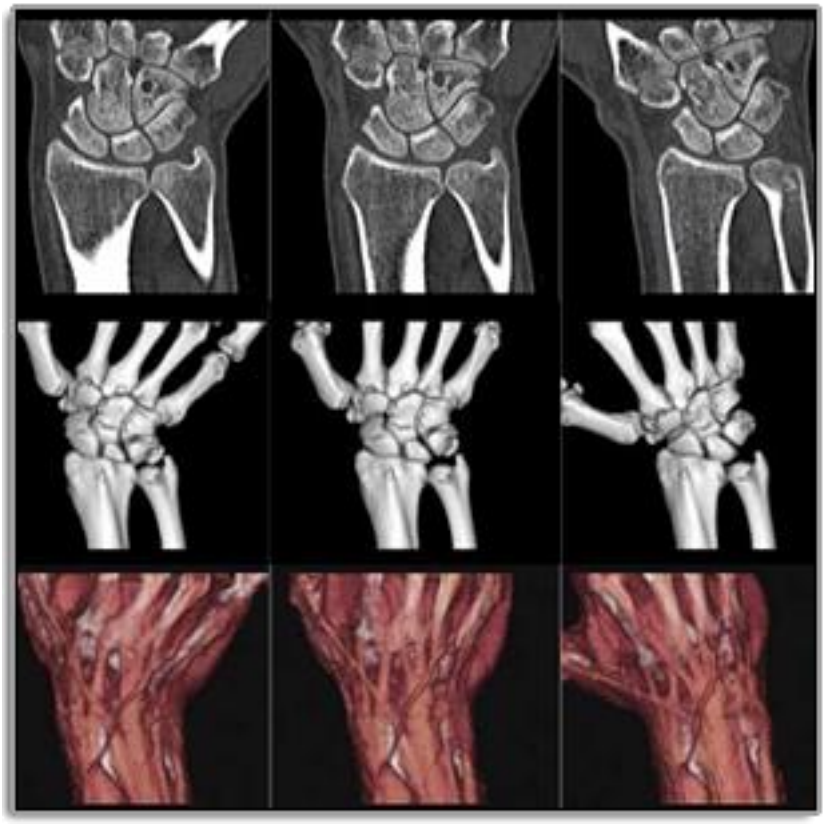


Figura 44: Escáner de diferentes capas de la mano derecha.

La propuesta futura de la aplicación de este proyecto con la ciencia médica radica en la capacidad de combinar sistemas en un método de aprendizaje quirúrjico orientado a cirujanos inexpertos.

Póngase el ejemplo de una apendicectomía que debe ser operada por un cirujano con experiencia. Esta operación con éxito sería grabada con la cámara TOF para almacenar los movimientos que ha realizado el cirujano sobre el paciente en concreto.

El paciente antes de ser operado se debería de someter a un escáner 4D para tener su modelo tetradimensional que se almacenaría como paciente modelo de operación de apendicectomía.

La utilidad estriba en que los cirujanos sin experiencia alguna podrían operar modelos virtuales basados en el modelo 4D del paciente, y dicha operación grabada con la cámara TOF se cotejaría con la experta intervención realizada por el cirujano experto que fue grabada en el momento real de la operación. De esta manera se consigue un entrenamiento puntuable por un sistema informático donde los recursos materiales se han sustituido por virtuales y donde el alumno puede probar tantas veces como sea necesario.

Por un lado se puede inducir que la presión que se ejerce sobre un cuerpo real con turgencia no es la misma que la que se ejerce sobre un modelo virtual, pero la aplicabilidad reside en la imitación de un experto, en seguir los pasos marcados o el protocolo creado para ello con más o menos acierto, no en la simulación del comportamiento físico del modelo virtual 4D como si fuera un cuerpo humano real.

Capítulo 6

Presupuesto

6.1 Claves

La claves para este capítulo son:

- Presupuesto del proyecto.
- Planificación del proyecto.
- Impacto del proyecto.

6.2 Introducción

El proyecto consiste en realizar una API clara y sencilla para el manejo de la cámara TOF y proponer una serie de aplicaciones que demuestran, por un lado el entendimiento y manejo de las características de la cámara, y por otro, las posibles finalidades de la misma.

A continuación se desglosa los costes imputados al proyecto y se presenta el diagrama Gantt de las tareas realizadas.

Presupuesto Costes Totales	Presupuesto Costes Totales
Personal	**19.303**
Amortización	**524**
Subcontratación de tareas	**0**
Costes de funcionamiento	**225**
Costes Indirectos	**4.010**
Total	**24.063**

Figura 45: Resumen Presupuesto.

Apellidos y nombre	Categoría	Dedicación (hombres mes)	Coste hombre mes	Coste (Euro)
Sánchez del Álamo Benguigui, Daniel	Ingeniero	1	4.289,54	**4.289,54**
Sánchez del Álamo Benguigui, Daniel	Investigador	2	4.289,54	**8.579,08**
Sánchez del Álamo Benguigui, Daniel	Programador	3	2.144,77	**6.434,31**
	Hombres mes 1		Total	**19.302,93**

Figura 46: Desglose costes de personal.

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable
PC	800,00	100	6	60	**80,00**
Camara TOF	3.500,00	100	6	60	**350,00**
Tripode	35,00	100	6	60	**3,50**
Visual Studio 2010 Proffesional	800,00	100	3	60	**40,00**
Excel 2010	189,00	100	1	60	**3,15**
Word 2010	189,00	100	1	60	**3,15**
Project 2010	775,00	100	1	60	**12,92**
PowerPoint 2010	189,00	100	1	60	**3,15**
Windows 7	285,00	100	6	60	**28,50**
				Total	**524,37**

Figura 47: Desglose costes materiales.

Descripción	Empresa	Costes imputable
Fungible	UC3M	**25,00**
Laboratorio	UC3M	**200,00**
	Total	**225,00**

Figura 48: Desglose costes directos del proyecto.

Los costes medios mensuales oscilan en torno a cuatro mil euros, teniendo en cuenta que cerca del 80% del presupuesto es derivado de los costes de personal.

El presupuesto total de este proyecto asciende a la cantidad de VEINTICUATRO MIL SESENTA Y TRES EUROS (24063,00 €).

Colmenarejo a 17 de Septiembre de 2010

El ingeniero proyectista

Fdo. Daniel Sánchez del Álamo Benguigui

Capítulo 6

Glosario

API	*Application Programming Interface*
GDI	*Graphics Device Interface*
GUI	*Graphic User Interface*
ID	*IDentifier*
IDE	*Integrated Development Environment*
MVC	*Modelo-Vista-Controlador*
MVVM	*Model-View-ViewModel*
.NET	*Framework de Microsoft para desarrollar aplicaciones*
PC	*Personal Computer*
TOF	*Time-Of-Fligth*
USB	*Universal Serial Bus*
UI	*User Interface*
WPF	*Windows Presentation Foundation*
XACT	*Cross-platform Audio Creation Tool*
XAML	*eXtensible Application Marked Language*
XBOX	*Consola de Microsoft*
XML	*Extensible Marked Language*

Referencias

1. **Microsoft.** MSDN - Namespace. [En línea] http://msdn.microsoft.com/es-es/library/z2kcy19k(VS.80).aspx.
2. —. MSDN - Ensamblados. [En línea] http://msdn.microsoft.com/es-es/library/8wxf689z.aspx.
3. —. MSDN - Arquitectura WPF. [En línea] http://msdn.microsoft.com/es-es/library/ms750441.aspx.
4. —. MSDN - HLSL. [En línea] http://msdn.microsoft.com/en-us/library/bb509561(v=VS.85).aspx.
5. **OpenGL.** OpenGL - GLSL. [En línea] http://www.opengl.org/documentation/glsl/.
6. **Nvidia.** Nvidia - CG. [En línea] http://developer.nvidia.com/page/cg_main.html.
7. **Creator's Club XNA.** Creator's Club XNA. [En línea] http://creators.xna.com/en-US/.
8. **TOF Camera Brands.** TOF Camera Brands. [En línea] http://en.wikipedia.org/wiki/Time-of-flight_camera.
9. **CodePlex.** CodePlex - Fluent. [En línea] http://fluent.codeplex.com/.
10. **XBOX.** XBOX - Natal. [En línea] http://www.youtube.com/user/xbox.
11. —. XBOX - Kinect. [En línea] http://www.xbox.com/es-ES/kinect/.
12. **Mesa Imaging.** Mesa Imaging. [En línea] http://www.mesa-imaging.ch/.
13. **Dadlani, Sanyu Melwani.** *Detección de Posturas desde Modelos 3D.*
14. **Christian Lange, Thomas Hermann, and Helge Ritter.** *Holistic Body Tracking for Gestural Interfaces.*
15. **Hiroshi Arisawa, Takako Sato and Takashi Tomii.** *Human-Body Motion Simulation Using Bone-Based Human Model and Construction of Motion Database.*
16. **Edmée Amstutz, Tomoaki Teshima, Makoto Kimura, Masaaki Mochimaru and Hideo Saito.** *PCA-based 3D Shape Reconstruction of Human Foot Using Multiple Viewpoint Cameras.*
17. **P. Cerveri, M. Rabuffetti, A. Pedotti and G. Ferrigno.** *Real-time Human Motion Estimation Using Biomechanical Models and Non-linear State-space Filters.*
18. **Christian Theobalt, Edilson de Aguiar, Marcus A. Magnor and Hans-Peter Seidel.** *Reconstructing Human Shape, Motion and Appearance from Multi-view Video.*

19. **Thierry Oggier, Michael Lehmann, Rolf Kaufmann, Matthias Schweizer, Michael Richter, Peter Metzler, Graham Lang, Felix Lustenberger and Nicolas Blanc.** *An All-Solid-State Optical Range Camera for 3D Real-time Imaging with Sub-centimeter Depth Resolution.*
20. **Steffen Knoop, Stefan Vacek and Rüdiger Dillmann.** *Sensor Fusion for 3D Human Body Tracking with an Articulated 3D Body Model.*
21. **Miguel Cazorla, Diego Viejo and Cristina Pomares.** *Study of the SR4000 Camera.*
22. **Springer.** Springer. [En línea] http://www.springer.com/?SGWID=0-102-0-0-0.
23. **Philips.** Philips - iCT Brilliance. [En línea] http://www.healthcare.philips.com/us/products/ct/products/ct_brilliance_ict/index.wpd.
24. **Nvidia.** Nvidia - PixelShader. [En línea] http://www.nvidia.com/object/feature_pixelshader.html.
25. —. Nvidia - Shader. [En línea] http://developer.nvidia.com/object/fx_composer_home.html.
26. **Microsoft.** MSDN - XACT. [En línea] http://msdn.microsoft.com/en-us/library/ff827590.aspx.
27. —. MSDN - XAML. [En línea] http://msdn.microsoft.com/es-es/library/ms752059.aspx.
28. **Creator's Club XNA.** XBOX - RacingGame. [En línea] http://exdream.com/XnaRacingGame/.
29. **Microsoft.** Microsoft - Visual Studio. [En línea] http://emea.microsoftstore.com/es/es-ES/Microsoft/Diseno-+-Desarrolladores/Visual-Studio-2010?WT.mc_id=pointitsem_visualstudio_generic_2010&WT.srch=1.