Universidad Complutense de Madrid
Facultad de Informática
Sistemas Informáticos 2005 -2006

Memoria del Proyecto

Dispositivo de interfaz humana (HID) basado en Motion Capture Óptico

Profesor Director:
José Antonio López Orozco

Silvia Carvajal Romero
Olmo del Corral Cano
Orión García Gallardo

Índice

1. Objetivos

El proyecto consiste en la realización de un sistema de captura de movimientos basado en puntos de luz y cámaras, y su utilización como interfaz desde la cual un usuario puede interactuar con el ordenador.

Desde nuestro punto de vista, el siguiente paso en la evolución de los dispositivos de comunicación entre usuario y ordenador pasa por, entre otras cosas, la generalización del concepto de ratón a las tres dimensiones, permitiendo además varios punteros simultáneos.

Con estas capacidades de comunicación se puede posibilitar la aparición de una nueva rama de aplicaciones con un uso más intuitivo, mayor productividad y una mejor experiencia para el usuario.

La tecnología principal en la que un dispositivo de éstas características podría basarse sería la Captura de Movimiento (Mocap) Óptica, pero su principal desventaja es su elevado coste. Si queremos que la implantación de este tipo de dispositivos sea una realidad para el usuario común es necesario que existan soluciones mucho más baratas.

Teniendo en cuenta éste objetivo, nuestra solución utilizará dispositivos hardware económico y fácil de adquirir, como son las cámaras WebCam Usb y basará la mayor parte de su funcionamiento en software.

1.1. Resumen en castellano

El proyecto consta de tres etapas:

- Primera fase se diseñará e implementará los algoritmos necesarios para capturar la localización de puntos de luz en WebCams comunes y la posterior reconstrucción de las coordenadas tridimensionales de estos puntos en tiempo real, para así obtener un ratón 3D multipuntero que facilite el tratamiento de aplicaciones inherentemente tridimensionales.
- Segunda etapa, una vez conseguido la reconstrucción de puntos de luz, se intentará enriquecer el API de este dispositivo posibilitando distintas funciones como Reconocimiento de Gestos, o aumentar la eficacia del proceso de reconstrucción 3D basándolo en esqueletos.
- Finalmente se desarrollará un número variable de programas demostrativos que utilicen esta tecnología, como por ejemplo: Selector de Color RGB, Simulación de posicionamiento y dirección de una cámara en un universo 3D, ó Driver de Ratón 2D para Windows.

El lenguaje de implementación será C# gracias a su productividad, a la interoperatibilidad con C++ para el desarrollo de Drivers, y a la integración con Direct3D y Windows de la plataforma .Net, simplificando el proceso de desarrollo de Servicios de Windows o demos 3D.

1.2. Resumen en inglés

Our project is developed in three stages:

- In the first stage we will design and implement the needed algorithms for capturing light dots with ordinary WebCams, and then reconstruct the 3D coordinates in real time. The result will be a 3D mouse with multi-point capabilities, giving us a new way of using 3D applications.
- On the second stage, once we have captured the 3D information of the points, we will try to enrich the device's API by allowing new possibilities like Gestures Recognition, or increasing the reliability of the 3D reconstruction algorithm using bones.
- Finally, we will develop some demonstrative applications that use our technology, for example: An RGB colour selector, a simulator of camera position and orientation in a 3D universe, or a Mouse Device Driver.

We have chosen C# as the implementation language because its productivity, interoperability with C++ for device driver development, and integration with Direct3D and Windows of the .Net platform, simplifying a Windows Service or 3D demos development.

2. Introducción

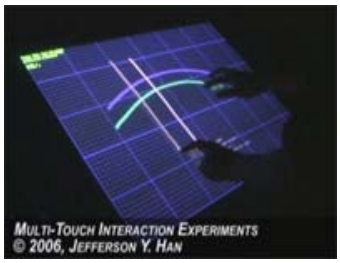
Las alternativas posibles que tiene un usuario a la hora de elegir los dispositivos para manejar su ordenador es un campo de la informática que avanza a un ritmo menor que otras ramas de esta ciencia.

Hace algo más de una década que los dispositivos de realidad virtual amenazan con la entrada en el mercado pero aún no han acabado de consolidarse. Al igual que en 1985, el usuario común de un PC continúa usando su ordenador con un teclado y un ratón, que fue inventado hace 38 años. Por otro lado, los usuarios de videoconsolas siguen manejándolas con versiones mejoradas de gamepads o joysticks que fueron inventados antes que la informática para el control de aviones.

Sin embargo, parece que comienzan nuevas iniciativas en el campo de los dispositivos de interfaz humana:

- La empresa de videojuegos Nintendo ha apostado por su misterioso mando tridimensional en su nueva videoconsola: Nintendo Wii.

- Un investigador de Apple Computer ha sorprendido a todos con una pantalla multitáctil. Apple ha patentado la idea.



- El nuevo sistema operativo de Microsoft, Windows Vista generaliza sus ventanas a 3 dimensiones requiriendo de tarjetas aceleradoras para su funcionamiento.

- Logitech ha comprado 3DConnexion empresa líder en la creación de dispositivos para el control de aplicaciones tridimensionales.

2.1. Dispositivos similares

Actualmente, existen muchas alternativas en el mercado en cuanto a dispositivos de interfaz humana se refiere. En el Anexo 1 repasamos los dispositivos de interfaz humana tradicionales y algunas de las nuevas ideas.

Es importante destacar que ningún 'Dispositivo de Interfaz Humana' de los explicados en el Anexo 1 es capaz de proporcionar al ordenador un dispositivo multipuntero 3D.

El dispositivo que más se aproxima es el Motion Capture Óptico, pero la desventaja principal son sus elevados costes, que rondan los 100.000€. Además, dicho dispositivo está enfocado a capturar movimiento en 3D, no a interactuar con el ordenador.

Para rebajar drásticamente este precio utilizaremos dispositivos hardware económicos y fáciles de adquirir, como son las WebCam Usb.

Como hemos dicho, hoy por hoy la tecnología que mejor puede satisfacer estos requisitos es la que utilizan los sistemas de Motion Capture Ópticos:

- **Precisión**: de todas las tecnologías de Motion Capture, la Óptica es la más precisa.
- **Versatilidad**: dispositivo configurable por software sin acoplarse a ninguna fisonomía ni funcionamiento particular.

2.2. Tecnologías

Antes de comenzar a realizar cualquier tipo de diseño de la solución, debemos partir de una elección de las posibles tecnologías existentes, tanto en hardware como en software.

En nuestro caso, la elección del hardware se trata fundamentalmente de la elección de dispositivos de captura. Podíamos elegir entre Tarjetas Capturadoras y WebCams, siendo ambas posibilidades relativamente baratas (la opción de Tarjetas Capturadoras es de un precio más elevado, junto con otros inconvenientes que detallamos en el punto 2.2.1.1. Tarjetas capturadoras).

Aunque finalmente elegimos las WebCams, debíamos también seleccionar una de entre todas las disponibles (ver selección en el punto Anexo 2: WebCams), basándonos en sus características y su precio. Esta elección es complicada y la variamos a lo largo del proyecto. Afortunadamente estos cambios no afectaron al código.

En cuanto al software, queríamos desarrollar una aplicación al alcance de todos, realizada con el sistema operativo más extendido, Windows, y la tecnología líder desarrollada para éste: .Net Framework. Puesto que esta plataforma cumplió las expectativas de rendimiento, donde teníamos mas dudas, no probamos otras alternativas.

A continuación estudiaremos más detenidamente estas elecciones:

2.2.1. Recursos Hardware

La elección del hardware es importante, pues influye directamente en la solución: dependiendo del hardware utilizado puede aumentar o disminuir la calidad del proyecto, o incluso no funcionar la aplicación.

Afortunadamente, gracias a la abstracción que da DirectShow, es posible cambiar los dispositivos hardware elegidos sin modificar el código.

2.2.1.1. Tarjetas capturadoras

Las tarjetas capturadoras capturan video y audio en tiempo real al disco duro del ordenador y en formato televisivo (PAL 768X576, DV 720X576 y a 25 fps).

Dichas tarjetas están especialmente diseñadas para sistemas de seguridad. Puede ser utilizada para multitud de usos, seguridad en tiendas, monitorización de exteriores, seguridad en edificios, vigilancia en estaciones de servicio, seguridad doméstica, etc.

Las cámaras de video analógicas trabajan realizando la captura de imágenes de forma **entrelazada**. No muestran toda la imagen en un mismo fotograma, sino que comienza a aparecer en las líneas superiores y sucesivamente se van rellenando el resto hasta

llegar a las líneas inferiores. Un único fotograma no es mostrado "de golpe", sino de modo secuencial. Este proceso de actualización de líneas es tan rápido que, en principio, a nuestro ojo le pasa desapercibido y lo percibimos todo como un continuo. Para que cuando se actualicen las líneas inferiores no se desvanezcan las superiores, dividimos las líneas en dos campos: uno formado por las líneas pares y otro formado por las impares.

Hay dos tipos de tarjetas capturadoras: las que capturan por software y las que lo hacen por hardware.

- *Capturadoras por software*: la CPU procesa todo el trabajo, tanto si capturamos con compresión como si lo hacemos sin ella. La tarjeta llevará un CD de instalación con los drivers correspondientes y un programa para la captura y edición de los vídeos. Cuando estemos capturando o editando la CPU se verá, por tanto, sobrecargada.

- *Capturadoras por hardware*: también conocidas como tarjetas de edición. Presentan la ventaja de poder capturar video con gran calidad y sin ocupar mucho espacio gracias a la compresión en tiempo real que realizan, además hacen prácticamente todo el trabajo liberando a la CPU y aceleran la exportación final del video. Suelen conectarse al ordenador por puerto USB o Firewire las externas, o por PCI las internas.

En un principio barajamos la posibilidad de tarjetas capturadoras en contraposición al uso de WebCams. Pensamos en utilizar una tarjeta capturadora PCI y 4 cámaras analógicas a color (con gran calidad) o tipo 'bullet' (más pequeñas y manejables).

Finalmente ésta posibilidad fue descartada por los siguientes motivos:

- Son más caras.
- Se utilizan en ordenadores de sobremesa,
- No suelen poseer control de exposición, lo que nos impide obtener una imagen totalmente oscura con únicamente los puntos de luz visibles. Esta última desventaja es muy importante, pues debemos trabajar en escenas lo más oscuras posibles, siempre que las luces que exponemos sean visibles (ver más ampliado punto '**3.2.2.2.** Control de tiempo de exposición').

2.2.1.2. WebCams

Un dispositivo WebCam consiste en una cámara, por lo general, de no muy alta calidad, que se conecta al ordenador por un puerto USB. Su utilización principal es para hacer videoconferencia por Internet. El cuello de botella se encuentra por tanto en la transmisión de video por la red, lo que ha hecho que las cámaras web no evolucionen demasiado durante estos años. Tienen, sin embargo, un bajo coste debido a su gran popularidad.

De todas las opciones barajadas, decidimos apostar por: **Logitech Quickcam Comunicate STX.**

Descripción del producto	Logitech Quickcam Comunícate STX Plus
Tipo sensor óptico	CMOS
Resolución video	640x480
USB	1.1
Ángulo	42º
Driver múltiple	si
Precio	50-70€

Cómo se puede observar, éste modelo dispone de driver múltiple: la capacidad de capturar imágenes simultáneamente de varias cámaras iguales. Evidentemente, éste es un requisito imprescindible para nuestro proyecto.

Ya avanzado el proyecto, utilizamos también otro modelo de WebCam, **USB Chatcam2**, pues ésta también posee driver múltiple (condición necesaria), y además posee mayor control del tiempo de exposición.

Otras valoraciones positivas de menor peso es su precio (sensiblemente menor) y su flexibilidad a la hora de adaptarla al portátil pues posee un soporte para pantalla TFT.

Descripción del producto	USB Chatcam2
Tipo sensor óptico	CMOS
Resolución video	640x480
USB	1.1
Ángulo	42º
Driver múltiple	si
Precio	35-45€

Se puede consultar el significado de las características de las cámaras en el apartado Anexo II: Estudio de WebCams.

2.2.1.2.1. Características del USB

- Un sistema USB tiene un diseño asimétrico, que consiste en un solo servidor y múltiples dispositivos conectados en una estructura de árbol utilizando dispositivos hub especiales. Se pueden conectar hasta 127 dispositivos.

- El diseño del USB elimina la necesidad de adquirir tarjetas separadas para poner en los puertos PCI, y mejorar las capacidades plug-and-play permitiendo a esos dispositivos ser conectados o desconectados al sistema sin necesidad de reiniciar.

- Los hubs USB 1.1 disponen de un ancho de banda de 12 Mbit/s debiéndose repartir entre todos los dispositivos (USB 1.1) conectados.
  Los hubs USB 2.0 disponen de un ancho de banda de 480 Mbit/s, que, de la misma manera, se han de repartir ente todos los dispositivos (USB 2.0) conectados.
  Sin embargo, al conectar dispositivos USB 1.1 a hubs USB 2.0, generalmente no se reparten los 480 Mbit/s del hub USB 2.0, sino los 12 Mbit/s del USB 1.1. Esto es debido a que la inserción de cada dispositivo dentro del flujo es responsabilidad de unas unidades especiales internas al hub denominadas **transaction translators** (traductores de transacciones) que no puede trabajar con varios dispositivos simultáneos. El TT reconoce automáticamente que tipo de dispositivo está conectado a cada puerto, y traduce cualquier señal USB 1.1 a USB 2.0, pero al haber un único TT para todos los puertos, el ancho de banda disponible para todos los dispositivos USB 1.1 sigue siendo de 12 Mbit/s.

  Se puede evitar este problema si se emplean hubs de mayor calidad, que dispongan de tantos TTs como puertos.

  Sin embargo no hemos encontrado estos hubs **Multi-TT** en el mercado.

Ventajas:
Ya que el estándar USB fue pensado para ser un interfaz para distintos tipos de periféricos, se decidió que los dispositivos se conectarían a través del hubs USB, así el número de puertos disponibles se ve aumentado pudiéndose conectar más cantidad de dispositivos que número de conectores disponibles en el ordenador.

Inconvenientes:
El ancho de banda del puerto se divide entre el total de dispositivos conectados, lo que puede dar problemas cuando se emplean dispositivos que consuman un gran ancho de banda.

Para ajustar el ancho de banda que cada dispositivo necesita los dispositivos USB pueden acceder a través de cuatro posibles sub-protocolos:

- *bulk*: para la mayoría de los dispositivos externos de almacenamiento.
- *control*: el 10% del ancho de banda está reservado para el protocolo de control, que dirige todas las transferencias.
- *Interrupt*: para teclados y ratones USB. Permite al ordenador saber cuando se produce un evento, como una pulsación de tecla o movimiento de ratón, sin necesidad de estar comprobando periódicamente el puerto USB.
- *isocronous*: las webcams trabajan en modo isócrono ("en tiempo real"), porque necesitan siempre un mínimo de ancho de banda para ellos. Siempre que se conectan dos dispositivos USB que trabajen con el protocolo isócrono y necesiten una cierta cantidad de ancho de banda, quedará muy poco disponible para el resto de los dispositivos. USB emplea el sistema FCFS (primero en llegar, primero en salir), por lo que si conectamos un tercer o cuarto dispositivo al hub, es probable que el sistema no disponga de suficiente ancho de banda para que funcione.

2.2.1.2.2. Control de tiempo de exposición

En fotografía, la exposición es la cantidad total de luz que cae sobre una película (o un sensor electrónico en caso de fotografía digital).

Todas las cámaras permiten variar el tiempo de exposición: el tiempo en que el obturador está abierto permitiendo entrar la luz. Cuando el tiempo de exposición es pequeño las imágenes son más oscuras, mientras que cuando es grande las imágenes son más claras y se aprecia el efecto 'motion blur'.

En nuestro caso, queremos que las cámaras capturen puntos de luz artificiales debiendo eliminar al máximo cualquier tipo de luz natural e intentando que la luz indirecta sea mínima. Para ello, debemos bajar al máximo el tiempo de exposición, para que así capture únicamente los puntos de luz, obviando todo tipo de destello y siendo también el ambiente de la escena lo más oscuro posible.

El resultado ideal, es una imagen totalmente oscura con únicamente los puntos de luz visibles.

Es muy importante que se cumpla dicho entorno al ejecutar cualquier tipo de prueba con nuestra aplicación, pues al trabajar con imágenes y manipularlas, éstas deben ser interpretadas obteniéndose puntos (nuestra finalidad principal) y cuánto más fácil nos resulte encontrar los puntos de luz, más rápido será el algoritmo y más fiables los datos para las fases posteriores.

Debido a la importancia de este punto explicaremos más extensamente la necesidad de un tiempo bajo de exposición.

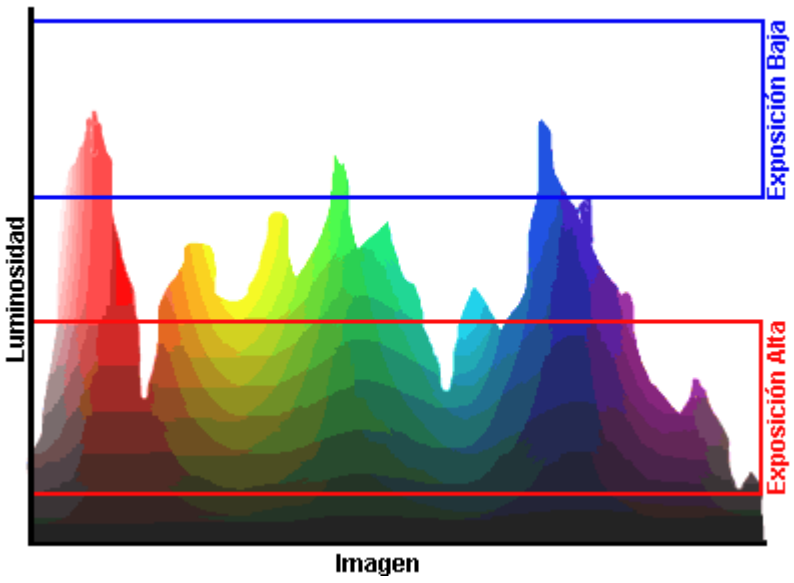
He aquí la diferencia entre una imagen con alta exposición y otra con baja exposición:

En la imagen con baja exposición, los puntos de luz son claramente diferenciados, mientras que con una alta exposición tanto los puntos de luz como el entorno, son de color análogo incrementando la dificultad de identificación de puntos para un algoritmo.

Podíamos haber utilizado filtros para solucionar dicho problema, pero esta opción contiene los siguientes inconvenientes:

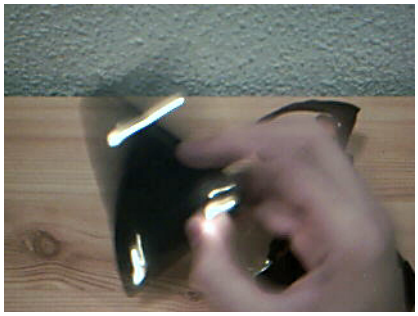
1- En el mundo real hay muchos niveles de brillo, pero al discretizar una imagen con un tiempo de exposición fijo, ésta se queda a un subconjunto de la luminosidad posible. Cualquier objeto más brillante será considerado blanco puro (255), y cualquier objeto más oscuro será negro (0).

Si sobre esta imagen discreta aplicamos filtros, podremos *jugar* con los valores de luminosidad de la imagen, pero nunca podremos diferenciar entre dos tonalidades que se salgan del intervalo de luminosidad capturado, por ejemplo: Una camiseta blanca (255) no podrá diferenciarse del sol (255) utilizando solamente filtros digitales.



Además, también necesitaríamos realizar cambios por software para filtrar las imágenes, produciéndose una pérdida del rendimiento.

2- Si conseguimos bajar el tiempo de exposición y mantener el objetivo de la cámara menos abierto disminuiría el número de "gusanos" introducidos por la misma. Podemos observar la citada diferencia entre estas dos imágenes:



3- Si conseguimos bajar el tiempo de exposición, aumentaríamos los frames por segundo de dicha imagen, pues cada imagen se captura más rápido, mejorando el resultado global de la aplicación

Por lo tanto, si bajamos el tiempo de exposición, la cámara captará las luces blancas del entorno de una forma nítida, desaparecerán los gusanos, y el sistema irá a mayor velocidad.

2.2.1.2.3. Alternativas de productos

A la hora de elegir las WebCams consideramos las siguientes características:

- ***Tipo sensor óptico***

El sensor sensible a la luz es un chip CDD (Dispositivo de Carga Acoplada) o CMOS (Semiconductor Complementario de Metal-Óxido). Ambos emplean como principio básico para la obtención de la imagen el efecto fotoeléctrico, en el cual la incidencia de luz sobre un cierto metal produce una intensidad de corriente eléctrica directamente proporcional a la intensidad de ésta. La superficie de ambos tipos de chips es un conjunto de pequeñas células fotosensibles, llamadas píxeles o canales, que captan la luz y generan la carga eléctrica.

La principal diferencia técnica entre CDD y CMOS, radica en la transmisión y conversión de la señal analógica. Mientras que el CDD debe transportar la señal hasta los extremos del sensor para amplificar y transmitirla, en el CMOS cada píxel incorpora un amplificador de la señal eléctrica y el conversor digital se encuentra integrado en la propia estructura del sensor.

Los CMOS son semiconductores, por lo que requieren un proceso de fabricación menos complejo, más económicos, y un consumo de energía por debajo de los CDD.

Aunque en principio, los CDD por el momento dan mejor calidad de imagen que los CMOS, estos últimos nos han permitido más libertad para graduar la exposición.

- ***Resolución video***

Aunque la cámara pueda capturar las imágenes a una alta resolución (1024 x 768), debe reducirse debido a las limitaciones de ancho de banda del USB (640 x 480 o inferiores).
Casi todas las webcams capturan fotogramas a 30 frames por segundo, velocidad también limitada por el ancho de banda.
Por esta razón, la resolución de las imágenes y la frecuencia de refresco de las cámaras no ha sido una prioridad a la hora de elegir el modelo de WebCam.

- ***USB***

En principio USB 2.0 es mejor opción que USB 1.1. Sin embargo a día de hoy hay **muy pocas** webcams USB 2.0, aunque muchas, de manera engañosa, declaran ser *compatibles* con USB 2.0

- ***Ángulo***

El ángulo de visión debe ser lo más abierto posible.
En el mercado el de mayor amplitud disponible es un ángulo de 72º, aunque la mayoría son de 42º.

- ***Driver múltiple***

Entendemos por 'driver múltiple' la capacidad de un driver de funcionar con varias webcams del mismo modelo simultáneamente (con modelos distintos es más sencillo). Sorprendentemente, esta característica es muy poco común entre las webcams, y absolutamente vital para nuestro proyecto. Esto ha limitado nuestra libertad de elección para todas las características anteriores: la posibilidad de USB 2.0 y de ángulos mayores de 42 grados.

Ver comparativa en el Anexo 2: WebCams.

2.2.2. Recursos Software

Como en todo proyecto con una extensión significativa, es necesario, una buena documentación inicial, en la cual basarse para comenzar con buen pie las directrices del proyecto.

Debido a la utilización de componentes software reutilizables se consiguen las siguientes mejoras:

1. **Aumento de la productividad**.
   Pues podremos centrarnos en otros puntos importantes del proyecto diseñados e implementados por nosotros.
2. **Aumento de la calidad**.
   Hemos incrementado la calidad del proyecto. Al reutilizar componentes software en repetidas ocasiones, dichos componentes ofrecen garantías de que la 'pieza' carece de errores o 'fallos'. Cuanto más se reutiliza un componente menor es la probabilidad de encontrar errores en él.
3. **Disminución del tiempo de entrega**.
   Esto es una consecuencia de los dos puntos anteriores.

La reutilización de código existente en nuestro proyecto se encuentra detallada en el punto '2.2.2.2. Componentes Software reutilizados'.

2.2.2.1. Plataforma .Net

La elección de la plataforma de desarrollo nunca ha de ser una decisión que se tome a la ligera, pues es difícilmente reversible, limita tus opciones y determina considerablemente la calidad final de tu solución.

A pesar de que nuestro proyecto no encaja en ninguna categoría típica de software (a medio camino entre el desarrollo de videojuegos, drivers de dispositivos y software científico) los requisitos que debía cumplir la plataforma si lo eran: Productividad, y Eficiencia computacional del ejecutable.

El hecho de que el lenguaje Java no soporte tipos por valor, todas las llamadas sean llamadas virtuales, y no soporte sobrecarga de operadores hacen de ésta plataforma una mala solución para el uso intensivo que íbamos a hacer de estas características al utilizar Puntos, Matrices y Cuaterniones, pues empeoran el rendimiento y dificultan la legibilidad en algoritmos muy enfocados a las matemáticas.

La alternativa evidente era C++, pero unas primeras pruebas de rendimiento en el principal cuello de botella de la aplicación, el procesado de las imágenes obtenidas de las webcams en tiempo real, utilizando la plataforma .Net y, en concreto, código *unsafe* en C#, demostraron que ésta plataforma estaba a la altura en cuanto a los requisitos de rendimiento. Nos decidimos por tanto por ésta opción, pues tiene un entorno

(*framework*) más homogéneo y simple, y ofrece el conocido aumento de productividad de la gestión automática de memoria.

La segunda gran decisión respecto a la plataforma ha sido el API gráfica a utilizar. En la actualidad existen dos alternativas principales: OpenGL y DirectX.

OpenGL tiene a su favor ser algo más conocido por nosotros, sin embargo DirectX, al ser de Microsoft, dispone de una interfaz oficial desde .Net, Manager DirectX, más mantenida y utilizada que los *wrappers* no oficiales de OpenGL para .Net que existen en la actualidad: SharpGL (http://www.codechamber.com/sharpgl/home.php) y CsGL (http://csgl.sourceforge.net/).

A pesar del cambio, Managed DirectX ha resultado ser una grata sorpresa: Un API moderno, Orientado a Objetos, pero con todas las opciones que diferencian a una API gráfico de un motor gráfico: tener el control a bajo nivel sobre la tarjeta de video para hacer código eficiente.

Además, DirectX simplifica los aspectos más enrevesados de la geometría en OpenGL:

- Divide la matriz ModelView en dos, Modelado y Vista.
- Utiliza row-vector en lugar de column-vector, lo que lleva a trasponer todas las matrices de transformación, pero hace que las transformaciones se concatenen de izquierda a derecha y no al revés.
- Sigue un sistema de coordenadas Left-handled en lugar de Right-Handled, en la práctica, esto lleva a que el eje Z va hacia dentro de la pantalla y no hacia fuera, lo que homogeniza las matrices para la colocación de cámaras y luces, evitando la mayoría de las apariciones de coeficientes negativos.

2.2.2.2. Componentes Software reutilizados

La mayor parte del código que hemos reutilizado proviene de Microsoft .Net Framework 2.0 (tipos primitivas, colecciones genéricas, diseño de formularios…), y de la librería Managed DirectX en el que se encuentran las primitivas necesarias para hacer aplicaciones con mucho contenido en geometría 3D y de las que hemos hecho un uso intensivo.

Estas dos librerías han constituido la plataforma básica sobre la que se apoya la solución. Sin embargo, existen otras aportaciones en las que nos hemos basado o que directamente hemos integrado y adaptado:

- **Sharp3D**

http://www.ekampf.com/Sharp3D.Math/

Se trata de una librería de matemáticas para .Net escrita en C#. Ofrece tipos primitivos básicos como vectores, matrices, números complejos o cuaterniones, funciones de

análisis y funciones de distancia e intersección entre objetos geométricos como planos, rayos, esferas, etc…

En un principio utilizamos Sharp3D como base para todos nuestros algoritmos de carácter matemático, sin embargo a medida que Managed DirectX fue tomando relevancia en el proyecto comenzaron a haber demasiadas conversiones entre los tipos primitivos de Sharp3D y los de DirectX. Además, Sharp3D utiliza la filosofía de OpenGL en su funcionamiento: column-vectors y un sistema de coordenadas Right-Handled.

Decidimos por tanto homogeneizar el código y utilizar sólo los tipos primitivos de Manager DirectX, pues son más rápidos (al estar internamente programados en ensamblador) y más completos, portando los algoritmos y tipos más complejos de Sharp3D, como Rayos, Planos, que Manager DirectX no disponía. Este código se encuentra en nuestra librería SharpDirectX.dll.

- **Motion Detection Algorithms By Andrew Kirillov:**

http://www.codeproject.com/cs/media/Motion_Detection.asp

Managed DirectX no incluye DirectShow de manera oficial, por lo que no hay manera de crear un grafo de filtros para procesar video desde .Net, debiendo utilizar COM como se haría en C++.

En un artículo sobre Motion Detection en The Code Project escrito por Andrew Kirillov utiliza un conjunto de clases que controlan la captura de video, haciendo, además, que la captura de video a través de WebCams y archivos de Video cumplan la misma interfaz.

En nuestro proyecto hemos adaptado estas clases, limpiando toda la funcionalidad que no nos interesaba (Motion Detection) y añadiendo otra: La capacidad de configurar la webcam para que capture a una resolución determinada.

- **HadddEngine.AI.EvolutionaryComputing por Vicente Cartas Espinel**

http://www.haddd.com

La infraestructura básica para hacer un algoritmo genético es una modificación de la excelente librería de inteligencia artificial del motor para videojuegos Haddd para Managed DirectX. Ésta librería hace un uso intensivo de las características de C# 2.0 (*generics* y delegados) y está diseñada de una manera muy modular, de manera que es reutilizable para prácticamente cualquier problema de programación evolutiva.

Las modificaciones realizadas están enfocadas a mejorar el rendimiento, y permitir la ejecución de evoluciones paso a paso (generación a generación), pero gran parte ha sido utilizado dejándose intacto.

- **Quick Fill y Flood Fill Algorithm**

http://www.codeproject.com/cs/media/floodfillincsharp.asp
http://www.codeproject.com/gdi/QuickFill.asp

Nos basamos en las ideas de estos dos artículos de Code Project para desarrollar nuestro algoritmo de reconocimiento de puntos. De la primera aprendimos como leer un Bitmap utilizando punteros en C# (código *unsafe*). El segundo artículo, Quick Fill, implementa en C++ un algoritmo de 'varita mágica' muy eficiente, de él sacamos las ideas de cómo recorrer una imagen de manera eficiente y evitando errores en caché.

Sin embargo, no hay código reutilizado de estos dos artículos, simplemente nos sirvieron como base para desarrollar nuestro propio algoritmo.

3. Diseño e Implementación

En dicho apartado explicaremos extensamente el funcionamiento de nuestro proyecto. Sin embargo, antes de implementar cualquier solución software, es necesario determinar o acordar cual será su arquitectura y, en este caso particular es muy necesario explicar los conceptos matemáticos en los que se apoya.

3.1. Arquitectura

¿Qué es la arquitectura de software? Según la Wikipedia: Consiste en un conjunto de patrones y abstracciones coherentes que proporcionan el marco de referencia necesario para guiar la construcción del software para un sistema de información.
(http://es.wikipedia.org/wiki/Arquitectura_de_software**)**
Arquitectura es considerar un programa, no solamente como un conjunto de líneas de código, sino además como unas estructuras de más alto nivel que interactúan entre sí.
Cómo organizar tu código en estas estructuras depende de los *objetivos y reestricciones* de nuestro sistema software, que pueden ser funcionales y no funcionales.
Los objetivos que tiene que satisfacer son, en orden de importancia:

- **Rendimiento:** Es necesario que el proceso sea lo más rápido posible, pues el tiempo de respuesta influye por completo en la satisfacción del usuario.
- **Visibilidad:** La visibilidad es la capacidad de una arquitectura (y proceso) de mostrar resultados desde el primer día. Puesto que, el nuestro, se trata de un proyecto de investigación, teniendo por tanto un nivel de riesgo *muy* alto, es necesario poder ver el resultado de cada uno de los pasos del proceso a medida que se van construyendo.
- **Tolerancia a fallos:** Definir que ocurre cuando falla una cámara o está mal calibrada, desaparece un punto…
- **Facilidad de uso (API):** Utilizar nuestra solución desde una aplicación de terceros debe ser lo más sencillo posible.

De los requisitos anteriores, visibilidad y facilidad de uso son mejor atendidos por una visión más arquitectónica, a alto nivel, de la solución, que se explicará a continuación. El rendimiento y la tolerancia a fallos, sin embargo, tienen más sentido ser resueltos a más bajo nivel, desde el código y, sobre todo, los principios matemáticos, que estudiaremos en el punto '3.2 Principios Matemáticos'.

3.1.1. Arquitectura de componentes

En el apartado 3.4 Arquitectura comentábamos que la responsabilidad del modelo de componentes en nuestra práctica iba a ser satisfacer dos requisitos concretos: **visibilidad y facilidad de uso** por programas de terceros.

Si bien la facilidad de uso es un concepto más subjetivo y que puede resolverse, en última instancia, mediante la utilización de una fachada que resuma todo el funcionamiento del sistema, la visibilidad es un requisito más difícil de satisfacer, requiriendo de más atención en el diseño de la solución.

Hacer software visible, es hacer software por componentes, de manera que éstos se puedan probar independientemente del resto de la solución.

Según el concepto moderno de componente, y según **A**nders Hejlsberg, un componente tiene métodos, propiedades y eventos.
(http://en.wikipedia.org/wiki/Anders_Hejlsberg)
(http://www.artima.com/intv/simplexity.html)

Los eventos, juntos con las interfaces, son las herramientas principales con las que cuenta un arquitecto de software a nivel de código, pues son las que permiten *jugar* con el concepto de dependencia.

Se entiende por dependencia la relación que puede existir entre dos fragmentos de código en el que uno se apoya en otro para resolver su trabajo. En código esto se ve reflejado por:

Una clase A depende una clase B si:

- A es una subclase de B.
- A tiene un campo de tipo B.
- A tiene un método que referencia a B (por el tipo de retorno o parámetro).
- A utiliza un objeto de tipo B para resolver su trabajo (variable local).

O de una manera mas sencilla: A depende de B si B ha de compilar para que A lo haga.

En principio la dependencia no es un problema, pero lleva a sistemas mas acoplados, lo que trae consigo los siguientes problemas:

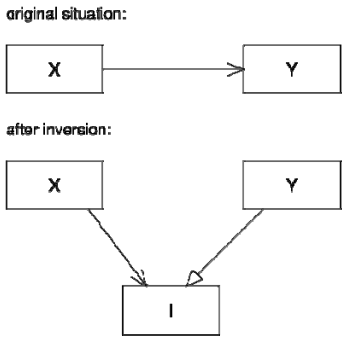
- Un cambio en una clase A produce una cadena de cambios en otras clases.
- Es difícil entender de manera aislada una clase.
- Es difícil reusar o probar de manera aislada una clase, pues necesita de otras para funcionar que también han de ser incluidas.

(http://en.wikipedia.org/wiki/Coupling_%28computer_science%29)

En la programación clásica, sin interfaces ni eventos, para hacer código bien estructurado, que pudiera ser entendido, probado y reutilizado con independencia del resto, era necesario sufrir la siguiente restricción: ¡La dependencia del código tenía que ir siempre en el mismo sentido que las llamadas! .

La programación orientada a objetos, con ideas como el polimorfismo, y en especial la implementación de interfaces (Java) y los eventos (Delphi y C#/.Net) tiene mecanismos para esquivar esta restricción. A ésta técnica se le llama Inversión de Control. (http://en.wikipedia.org/wiki/Inversion_of_Control).

En cierto modo, el problema que resuelve la Inversión del Control es el problema fundamental de la arquitectura de software y el que motiva la aparición de la mayoría de los patrones de diseño del GoF (Gang of Four, compuesto por Erich Gamma, Richard Helm, Ralph Johnson y John Vlisides).



La técnica clásica de hacer que una clase X llame a un método de otra clase Y sin depender de ésta es, desde java, que X conoce a Y a través de una interfaz I. De esta manera es el polimorfismo el que enrute la llamada del método de la interfaz al método concreto de la clase Y de manera dinámica en tiempo de ejecución.

En C#, y .Net en general, el problema de las dependencias se agrava pues, al igual que en Visual C++, existe el concepto de proyecto (exe, dll…) y no se permiten dependencias circulares entre proyectos. Existe en esta plataforma, además, el concepto de evento y delegado, que ofrece otra manera, más cómoda, de resolver la Inversión de control.

3.1.1.1. Delegados y Eventos:

Los delegados y los eventos son, tal vez, el concepto más complicado y controvertido de la plataforma .Net, y es utilizado intensivamente en nuestro proyecto, es por ello que dedicaremos un apartado a explicarlo en detalle.

Un delegado es un tipo, el tipo de las funciones que cumplen una firma, en concreto las que piden los mismos tipos en el mismo orden como parámetros y, tienen el mismo tipo de retorno. El delegado *por defecto* para muchos eventos de .Net es EventHandler, y está declarado como:

```
public delegate void EventHandler(object sender, EventArgs e);
```

Lo que indica que se trata del tipo de las funciones que reciben un object y un objeto EventArgs y no devuelven nada.

Un delegado puede asignarse a una función (estática o no):

```
EventHandler dela = new EventHandler(miObjeto.Funcion);
EventHandler delb = new EventHandler(MiClase.FuncionEstatica);
```

Puede ser llamado, lo que provoca que se llame a la función asociada:

```
dela(null, null);
```

Y concatenarse, generando una lista enlazada de funciones que han de ser llamadas consecutivamente:

```
EventHandler delab = dela + delb;
delab(null, null);
```

Con un campo y una propiedad de tipo EventHandler sería suficiente para hacer que nuestra clase fuera capaz de llamar (pasar el control de programa) a métodos de clases de las que **no depende,** resolviendo de una nueva manera el problema de la Inversión de Control.

Existen algunos detalles, sin embargo, que quedarían mal resueltos con esta solución:

- Si bien la gente debería suscribirse con los operadores de concatenación o suscripción (+=) y desubscripción (-=), nada evitaría que utilizaran el operador de asignación (=), eliminando de esta manera a cualquier otro suscriptor que hubiera.

  button1.Click = miClick; `//Asignación destructiva`

- De la misma manera, nada evitaría que desde fuera de la clase se invocara al delegado desde su propiedad.

  ```
  button1.Click(); //Llamada desde fuera
  ```

- Una Interface no puede tener campos, por tanto no podría tener un campo delegado.

Para solventar estos *cabos sueltos* se ha creado el concepto de evento, que no es más que una Propiedad especial para delegados, que en lugar de tener región get y set, tiene región add y remove:

```
//Código 1
private EventHandler click;
public event EventHandler Click
{
  add { click += value; }
  remove { click -= value; }
}
```

De esta manera, desde fuera, no es posible desuscribir a todos los eventos por una asignación equivocada, pues solo esta permitido añadir (+=) y quitar (-=). Tampoco es posible llamar desde el exterior.

Además, se puede poner un evento en una interface así:

```
//Código 2
public event EventHandler Click
{
  add;
  remove;
}
```

Finalmente, acabando éste pequeño paréntesis, existe una sintaxis resumida para los códigos anteriores, simplemente:

```
//Código 3
public event EventHandler Click
```

La sentencia anterior(Código 3) es automáticamente reemplazada por el compilador a: dentro de una clase, nos declarará el campo, e implementará el evento (Código 1) y dentro de una interface nos declarará el atributo dejando las regiones add y remove sin definir (Código 2).

3.1.1.2. Arquitectura Pipeline

Tanto los requisitos arquitectónicos, como la propia idea intuitiva del algoritmo general de Reconocimiento de Puntos llevan a una solución arquitectónica evidente: Un Pipeline.

Un pipeline, o tubería, consiste en un proceso secuencial que se aplica en etapas (stages) a unos datos, como si de una cadena de montaje se tratara.

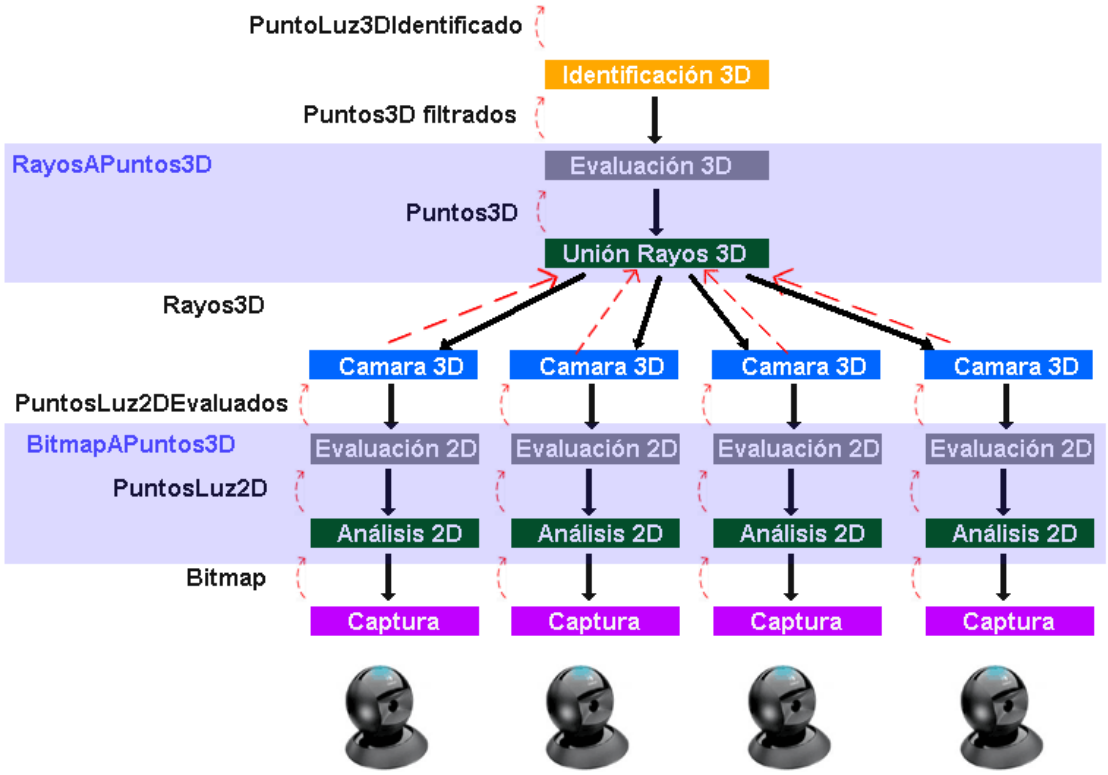
El pipeline es un tipo de arquitectura muy conocida, sobretodo en hardware, donde se aplica a procesadores o tarjetas gráficas. Desde éste punto de vista, y obviando muchos detalles, nuestro proyecto puede considerarse como el pipeline de una tarjeta gráfica recorrido de manera inversa.
(http://en.wikipedia.org/wiki/Graphics_pipeline)

A grandes rasgos el proyecto consiste en pasar las siguientes fases para cada cámara:

- Captura de la imagen desde las WebCam.
- Análisis de la imagen para extraer los Puntos2D.
- Evaluación de los Puntos2D, para filtrar los de baja calidad
- Proyección de los Puntos2D a Rayos3D

Y, una vez que todas las cámaras han llegado a este punto, sincronizar los hilos y reunir la información para que un solo hilo continúe el proceso:

- Mezcla de los rayos en 3D para generar Puntos3D
- Evaluación de los Puntos3D, para filtrar los de baja calidad.
- Identificación de los Puntos3D



En el gráfico se muestra la arquitectura en pipeline de nuestro proyecto, se aprecia cómo hay varios hilos para cada cámara, pero un solo hilo une el proceso. Así mismo, las flechas negras simbolizan las dependencias entre las clases (y proyectos), mientras que las flechas rojas simbolizan la dirección de la ejecución a través de los eventos.

Como podemos ver, todo el proyecto ¡es en si mismo un gran problema de Inversión de Control!, pues las unidades más simples (las primeras: captura de imagen, análisis en 2D…) han de ceder el control a las más complicadas, y no al revés.

Es por ello que casi todas estas etapas pasan el control de unos a otros a través de eventos, lo que permite, además, insertar escuchas a lo largo del proceso, pudiendo visualizar los resultados intermedios lo que facilita enormemente la depuración.

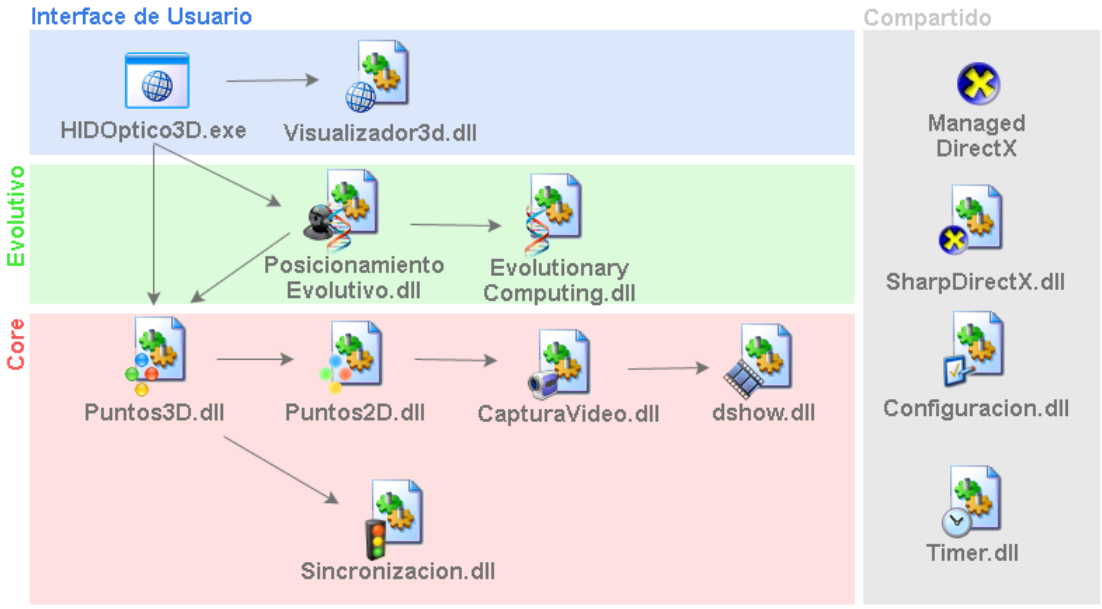
Gracias a esta arquitectura hemos podido visualizar resultados desde el primer momento (visibilidad), y probar las etapas de manera independiente (de las mas simples, a las mas complejas).

3.1.1.3. Componentes de la aplicación

Desde un punto de vista arquitectónico, el proyecto está dividido en diferentes componentes o, en la jerga de .Net, proyectos. Los principales tipos de proyectos que podemos encontrar en .Net son las DLLs y los EXEs. Ambos exponen código ejecutable que puede ser utilizado por terceros, pero los EXEs además definen un punto de entrada.

Los proyectos de nuestra aplicación se pueden dividir en cuatro grupos fundamentales que explicaremos a grandes rasgos:

- **Core:** Aquí se encuentran las DLLs principales necesarias para resolver toda la problemática del proyecto de una manera abstracta, desde la captura de imágenes hasta la identificación de Puntos3D. Se encuentra también la clase Loader3D, que unifica la manera de trabajar de la aplicación.
- **Evolutivo:** Dependiente del core, es el encargado de gestionar el algoritmo de posicionamiento evolutivo de las cámaras, útil para configurarlas cómodamente desde la interface de usuario.
- **Interface de Usuario:** Depende de las anteriores, y se encarga de configurar y visualizar de manera interactiva todo el funcionamiento de Core y Evolutivo. Se encuentra aquí HIDOptico3D.exe.
- **Compartido:** Éste es el grupo más heterogéneo, donde se encuentran las clases con funcionalidad más básica y de las que dependen los tres grupos anteriores. Managed DirectX, aunque no ha sido desarrollado por nosotros, también pertenece a esta categoría.



En el diagrama anterior pretenden mostrarse las dependencias **más importantes** entre los proyectos de cada grupo. Para hacerlo más legible no se han indicado, por ejemplo, las referencias a los proyectos de compartido, o aquellas que se pueden deducir de las indicadas: Si A refiere a B, y B a su vez refiere a C, entonces posiblemente A refiera a C pero no está indicado.

A continuación indicaremos las responsabilidades de cada unos de los componentes del diagrama, comenzando por la interfaz de usuario:

HIDOptico3D.exe: Es el *main* de la aplicación, contiene los formularios de la solución. Su misión principal es doble: permite configurar el sistema en un XML, y expone visualmente gran parte de su funcionamiento. Una explicación más detallada se encuentra en el apartado 4.Aplicaciones.

Visualizador3D.dll: Contiene los controles y clases necesarias para visualizar en 3D, haciendo uso de Managed DirectX, las distintas etapas del funcionamiento (y configuración) de HIDOptico3D. Estos controles se registran, directamente, a eventos lanzados por clases del Core, escuchando de manera transparente el proceso.

Como hemos dicho, el core es quien lleva acabo todo el proceso, y es el único grupo que es necesario incluir por una aplicación de terceros que quisiera incorporar HIDOptico3D.Estos son sus componentes:

Dshow.dll: Contiene las clases necesarias para la interoperatibilidad de .Net con DirectShow, que está en COM y no ha sido portada a .Net junto con el resto de DirectX (Managed DirectX).

CapturaVideo.dll: Esta segunda dll, utilizando dshow.dll, es la encargada de negociar la captura con las cámaras (configuración de la resolución), construir los bitmaps, y homogeneizar el interface para webcams y ficheros de video (avi, mpg…).

Puntos2D.dll: Analiza la imagen (Bitmap) capturada por las cámaras y reconocer las regiones de luz, anotando sus características. Se encarga también de evaluar estas características y filtrar todos aquellos puntos que no alcancen una nota determinada.

Sincronizacion.dll: Se encarga de sincronizar los hilos de las cámaras, de tal manera que no se dedique más procesador a una que a otra, y se lance el reconocimiento 3D hasta que no se hayan actualizado los datos para todas las cámaras.

Puntos3D.dll: Es la dll mas complicada. Genera los RayosLuz desde los PuntosLuz2D gracias a saber la posición y orientación de las cámaras y, ayudado por las clases de sincronización, une la información de todas las cámaras y la mezcla usando el algoritmo pirámide, generando así los Puntos3D. Finalmente evalúa e identifica los Puntos3D.

El grupo evolutivo no es necesario que se incluya en programas de terceros, pero si constituye una funcionalidad abstraída de la interface, que debe estar en dlls independientes, estos son sus componentes:

EvolutionaryComputing.dll: Versión personalizada de la librería de IA del motor HADDD. Contiene las clases de propósito general para el desarrollo de algoritmos genéticos.

PosicionamientoEvolutivo.dll: Concreta el sistema autocalibación de la cámaras, definiendo las operaciones de reproducción y mutación entre individuos de la clase PosicionCamara, que tienen una posición (punto) y una orientación (cuaternión).

Finalmente, el grupo de proyectos Compartido, agrupa a todas aquellas dlls de 'Utilidades' o con clases tan básicas que son utilizadas por casi cualquier componente del proyecto, Microsoft Managed DirectX también cumple estas características:
Managed DirectX: Estas dlls, desarrolladas por Microsoft, han sido incluidas dentro de este grupo porque contienen tipos básicos muy utilizados, como Vector3, Matrix o Cuaternion. Además son usadas por Visualizador3D.dll para la representación 3D del funcionamiento del sistema.

SharpDirectX.dll: Contiene el código fuente de la librería matemática Sharp3D, que no incluimos, sino que decidimos portar a las primitivas y el paradigma de Managed DirectX. Contiene por ejemplo la clase Ray, con la distancia entre dos rayos definidos, o la clase Plane.

Configuracion.dll: Gestiona todos los parámetros de configuración, desde la resolución de captura de las cámaras, a los parámetros de identificación de los puntos 3D, en clases estáticas de manera que son cómodamente accesibles desde cualquier parte. Define también las clases (no estáticas) para serializar estos datos a un fichero.

Timer.dll: Contiene la clase FastTimer, que permite tomar medidas de tiempo más precisas (por debajo de 15 MS).

3.1.1.4. Clases Principales

Antes de centrarnos en precisar las particularidades algorítmicas de las distintas fases del proceso de reconocimiento de puntos en 3D, es interesante ver una visión general sobre las clases que forman este proceso: El Core. El diagrama de la página siguiente muestra las clases fundamentales de éste grupo de dlls.

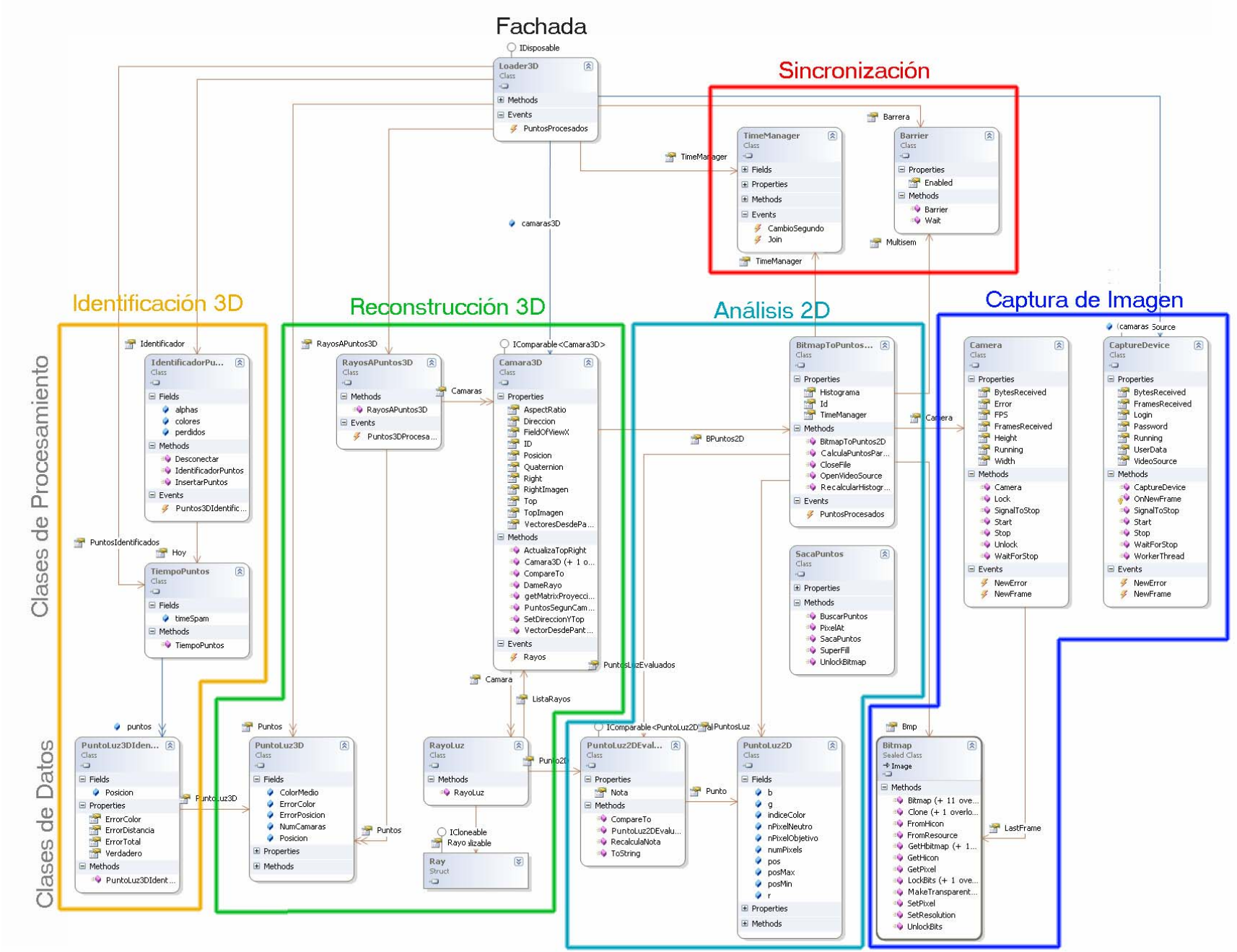
Está dividido verticalmente por las fases del *pepeline* del procesamiento de puntos: Captura de Imagen, Análisis 2D, Recostrucción 3D e Identificación3D.

Además, horizontalmente estas secciones están divididas en clases de procesamiento y clases de datos. Las clases de procesamiento permanecen con el ciclo de vida de la aplicación, y son las encargadas de transformar las clases de datos de un tipo a otro. Ésta es la razón por la que, en el diagrama, las clases de datos están ligeramente desplazadas a la izquierda, porque son el subproducto que se encuentra entre una clase de procesamiento y otra.

En este diagrama se puede observar, también, como las dependencias van desde las clases de mas alto nivel (Identificación 3D) a las de mas bajo nivel (Captura de imagen), y nunca al revés.

Sabemos, sin embargo, que la ejecución del código va en sentido inverso. Esto es posible porque lo hace a través de los eventos que tiene cada clase de procesamiento (con el icono debajo de cada clase).

Finalmente, en la parte superior se pueden ver las clases de sincronización y el Loader3D que actúa como fachada.


Fachada
Sincronización
Identificación 3D
Reconstrucción 3D
Análisis 2D
Captura de Imagen
Clases de Procesamiento
Clases de Datos
IDisposable
Loader3D
Class
Methods
Events
PuntosProcesados
TimeManager
Barrier
Properties
Enabled
Wait
Fields
CambioSegundo
Join
Multisem
camaras3D
Identificador
IdentificadorPu...
alphas
colores
perdidos
Desconectar
IdentificadorPuntos
InsertarPuntos
Puntos3DIdentific...
PuntosIdentificados
Hoy
TiempoPuntos
timeSpam
TiempoPuntos
RayosAPuntos3D
Puntos3DProcesa...
Camaras
IComparable<Camara3D>
Camara3D
AspectRatio
Direccion
FieldOfViewX
ID
Posicion
Quaternion
Right
RightImagen
Top
TopImagen
VectoresDesdePa...
ActualizaTopRight
Camara3D (+ 1 o...
CompareTo
DameRayo
getMatrixProyecci...
PuntosSegunCam...
SetDireccionYTop
VectorDesdePant...
Rayos
PuntosLuzEvaluados
Camara
ListaRayos
BitmapToPuntos...
Histograma
Id
BitmapToPuntos2D
CalculaPuntosPar...
CloseFile
OpenVideoSource
RecalcularHistogr...
BPuntos2D
SacaPuntos
BuscarPuntos
PixelAt
SacaPuntos
SuperFill
UnlockBitmap
Camera
BytesReceived
Error
FPS
FramesReceived
Height
Running
Width
Lock
SignalToStop
Start
Stop
Unlock
WaitForStop
NewError
NewFrame
CaptureDevice
FramesReceived
Login
Password
UserData
VideoSource
CaptureDevice
OnNewFrame
SignalToStop
Start
Stop
WaitForStop
WorkerThread
puntos
PuntoLuz3DIden...
Posicion
ErrorColor
ErrorDistancia
ErrorTotal
Verdadero
PuntoLuz3DIdent...
PuntoLuz3D
ColorMedio
ErrorColor
ErrorPosicion
NumCamaras
Posicion
RayoLuz
ICloneable
Ray
Struct
IComparable<PuntoLuz2D>
PuntosLuz
PuntoLuz2DEval...
Nota
CompareTo
PuntoLuz2DEvalu...
RecalculaNota
ToString
PuntoLuz2D
b
g
indiceColor
nPixelNeutro
nPixelObjetivo
numPixels
pos
posMax
posMin
r
Bmp
Bitmap
Sealed Class
Image
LastFrame

3.2. Principios matemáticos

Contar con un marco matemático completo en una solución como ésta, que hace un uso intensivo de geometría en 3D, es fundamental y es lo que diferencia nuestro proyecto de otras soluciones similares:

El proyecto ganador de la Imagene Cup 2006, Steb by step, excelente en otros aspectos no permite, sin embargo, el nivel de libertad y configurabilidad que permite nuestro proyecto. En su caso el sistema está limitado a dos cámaras en una posición fija. http://www.microsoft.com/spanish/msdn/estudiantes/eventos/imaginecup2006.asp

Es el *background* matemático del que estamos tan orgullosos, con primitivas como matrices de transformación, rayos, o cuaterniones, y con operaciones como la proyección, la distancia entre rayos o la *Spherical Linear intERPolation* el que confieren a nuestro proyecto ese nivel de libertad en parámetros muy concretos:

- **Número variable de cámaras:** Nuestro software permite utilizar un número variable de cámaras, siendo las soluciones prácticas entre 2 y 4 cámaras debido a problemas del ancho de banda del USB. Esta limitación no se encuentra por tanto en el algoritmo de reconstrucción de puntos 3D.
- **Posición y orientación arbitrarias:** En nuestro proyecto, cada una de las cámaras que utilicemos puede estar situada y orientada *hacia donde quiera*. Evidentemente existen mejores posiciones que otras, teniendo en cuenta factores como la oclusión o los problemas de precisión si dos cámaras están muy cerca pero, de nuevo, éste es un problema físico, no de nuestro algoritmo.
- **Características de la cámara:** ¡Permite utilizar distintos tipos de cámaras, con distinto *Aspect Ratio* y *Field of View*, y capturando a distintas resoluciones simultáneamente!
- **Auto Calibración:** Aunque no se trata de un grado de libertad por si mismo, es una herramienta completamente indispensable para permitir que las cámaras puedan posicionarse y orientarse arbitrariamente.

A continuación repasaremos los principios matemáticos en los que nuestro proyecto se ha apoyado para hacer posible las características anteriores:

3.2.1. Algebra y matrices

Dentro del álgebra, las matrices son el constructo matemático para las transformaciones de puntos en el espacio. Existen matrices para mover, rotar o escalar puntos.

Las matrices pueden ser concatenadas: en vez de aplicar una matriz de rotación, una de translación y otra para escalar a cada uno de los puntos del objeto, se puede obtener una sola matriz que hace todo esto en una operación, y así simplificar el cálculo, siempre teniendo en cuenta que el orden si es importante, al no poseer las matrices la propiedad conmutativa.

Existen además varias posibilidades a la hora de usar matrices: se puede considerar los puntos como columnas (aproximación más matemática) o como filas (aproximación más informática). La primera tiene como ventaja una notación más concisa en el papel, mientras que la segunda tiene la gran ventaja de que las transformaciones se aplican de izquierda a derecha, lo que es sin duda más natural. Ésta segunda alternativa es la que nosotros hemos utilizado.

Para un punto en 3D una matriz 3x3 podría parecer suficiente. Sin embargo la operación traslación no se puede representar en ésta forma matricial pues no es una combinación lineal de las 3 coordenadas. Surge por tanto el concepto de matrices y puntos **homogéneos**. Una matriz homogénea tiene un 4º vector que representa el desplazamiento del origen de coordenadas entre los dos sistemas de coordenadas. Y un punto homogéneo tiene una cuarta coordenada que solo debería ser 0 (para un vector) o 1 (para un punto).

$$\begin{pmatrix} Ux & Uy & Uz & 0 \\ Vx & Vy & Vz & 0 \\ Wx & Wy & Wz & 0 \\ Px & Py & Pz & 1 \end{pmatrix}$$

3.2.1.1. Matrices clásicas

1.- Traslación

Traslada un punto desde una posición (x, y , z) a otra posición (x',y',z') siendo dicha matriz:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Px & Py & Pz & 1 \end{pmatrix}$$

2.- Rotación

El eje de rotación siempre pasa por el origen, para rotar con un eje que no pase por dicho punto, basta trasladar el objeto antes de rotar, de forma que el origen corresponda al punto de rotación deseado. Una vez realizada la rotación puede realizarse una traslación inversa a la inicial para situar el objeto en la situación correcta. Hay que tener en cuenta que el orden en el que se realizan las rotaciones es importante.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & \sin(\alpha) & 0 \\ 0 & -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\alpha) & 0 & -\sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\alpha) & \sin(\alpha) & 0 & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(a) (b) (c)

(a) Matriz asociada a la rotación respecto al eje x
(b) Matriz asociada a la rotación respecto al eje y
(c) Matriz asociada a la rotación respecto al eje z

siendo α el ángulo de la rotación en todos los casos.

3-Escalación

El escalado consiste en multiplicar las coordenadas del objeto por una constante, para modificar su tamaño.
Se describe el escalamiento cuando el punto fijo es el origen.
Para obtener un escalamiento con un punto fijo arbitrario, este se debe trasladar al origen, escalar el objeto, y después realizar el inverso de la traslación original.
La inversa de un escalamiento se obtiene usando los recíprocos de los factores de escala 1/Ex , 1/Ey, 1/Ez.

$$\begin{pmatrix} Ex & 0 & 0 & 0 \\ 0 & Ey & 0 & 0 \\ 0 & 0 & Ez & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Con esta matriz de transformación, las longitudes en los ejes X, Y, y Z se multiplican respectivamente por Ex, Ey y Ez.

3.2.2. Matriz de proyección

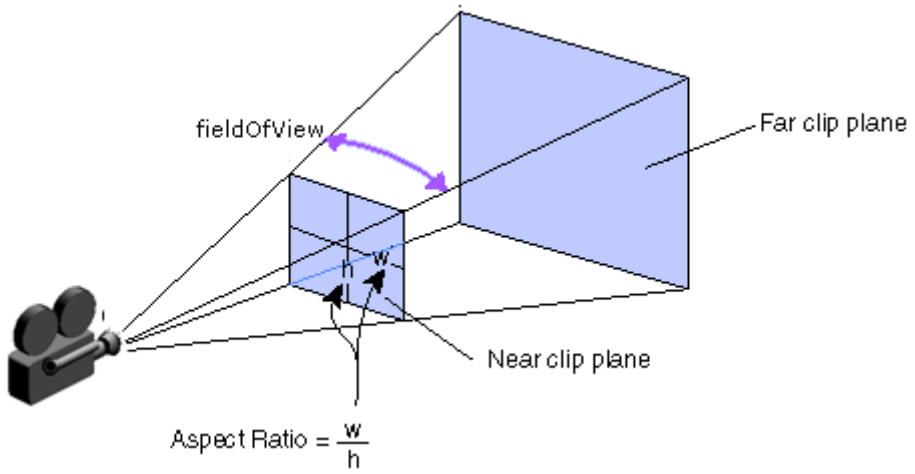
Una proyección de perspectiva es responsable de hacer que los objetos cercanos a la cámara se vean más grandes que los objetos en la distancia. Ejecuta una división para reducir y aumentar los objetos dependiendo de la distancia al observador. Es importante saber que las medidas de los objetos proyectados no tienen por qué coincidir con las del objeto real.

Se entiende por '*vista frustum*' la región visible por una proyección perspectiva, se trata de una pirámide con la cámara posicionada en la punta. Esta pirámide es intersecada por un plano de corte en el frente y en la parte trasera. El volumen dentro de la pirámide entre el plano de corte del frente y el plano trasero es la única región dónde son visibles los objetos.

Para construir dicha matriz de proyección debemos partir de:

- **Field of View** (FOV o campo de visión): se mide en grados y proporciona una medida de la extensión abarcada por la imagen.
- **Aspect ratio**: es la relación entre el lado más largo y el más pequeño (cociente entre el ancho de la pirámide dividido por el alto de la base de la pirámide).
- **Near clip Plane** (plano de corte cercano): distancia del plano de corte del frente.
- **Far clip Plane** (plano de corte lejano): distancia del plano de corte de atrás.

Todo lo que esté por delante del plano más cercano y todo lo que esté detrás del plano más lejano no será representado.



Dicha matriz por sí misma no realiza la proyección, pues no es posible realizar divisiones entre coordenadas, sólo combinaciones lineales, y como hemos comentado, los objetos han de verse más reducidos cuanto mayor sea su coordenada z.

Si intentamos solucionar esta ecuación, podremos observar como no encontramos ninguna M que cumpla que:

$$M * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x/(z/d) \\ y/(z/d) \\ d \\ 1 \end{pmatrix}$$

Una solución para realizar la proyección es acumular en la cuarta coordenada (w) el valor por el cual quiero dividir dicha matriz de proyección, para en un paso siguiente (fuera del álgebra de matrices) realizar dicha operación.

La operación explicada anteriormente sería de la forma:
Para cualquier w<>0, siendo w dicho valor almacenado.

$$p = (1/w) * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

3.2.3. Inversa de matriz de proyección

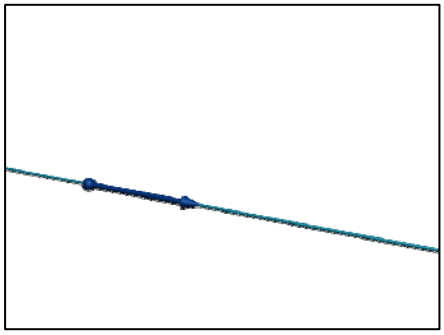
En el punto '3.2.2. Matriz de proyección' hemos explicado como dicha matriz, por sí misma, no realiza la proyección. Tal vez, deberíamos hablar más de '*matriz de proyección inversa'* que de inversa de matriz de proyección.

Existen transformaciones que no se pueden invertir, pues al realizar la primera transformación perdemos datos de la situación inicial.

Esta es una de las citadas situaciones, al realizar la primera transformación hemos perdido la coordenada z. Debido a esto, multitud de puntos se proyectan al mismo, en concreto todos los que pertenecen al rayo que pasa por el origen de la cámara y el punto proyectado.

Un rayo es una región mayor en espacio que un punto, es aquí dónde se percibe la pérdida de información. Podemos, sin embargo, recuperar esta información cruzando rayos de distintas cámara. Esta es la razón por la que la razón por la que tener dos ojos es útil para los humanos, y es la misma técnica que seguiremos nosotros en nuestro proyecto.

Un rayo está definido por un punto origen y una dirección que ha de estar normalizada.

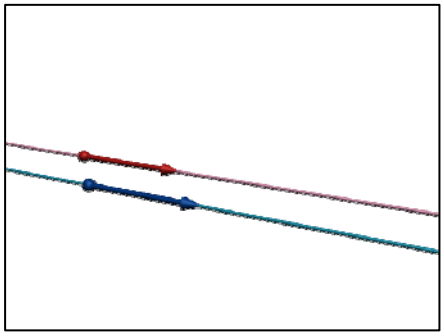


3.2.4. Distancia entre rayos

Puesto que, en la realidad, dos rayos nunca se cruzan sino que pasan muy cerca uno de otro, es necesario definir la distancia entre rayos.

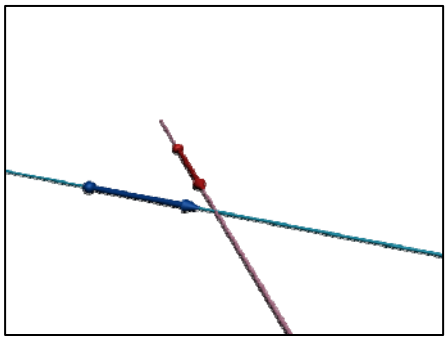
Para realizar este cálculo debemos diferenciar tres posibilidades:

- Los rayos son paralelos:



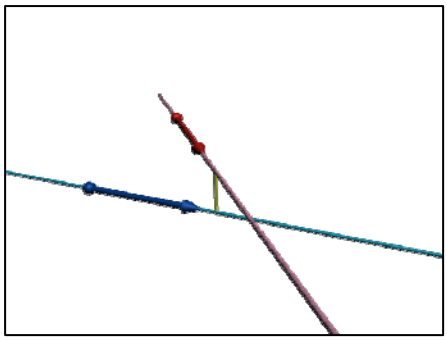
Direcciones iguales de los dos rayos.

- Los rayos se cruzan:



Los dos rayos están en el mismo plano, más concretamente, los orígenes de los dos rayos están en un mismo plano con orientación del vector perpendicular a las dos direcciones.

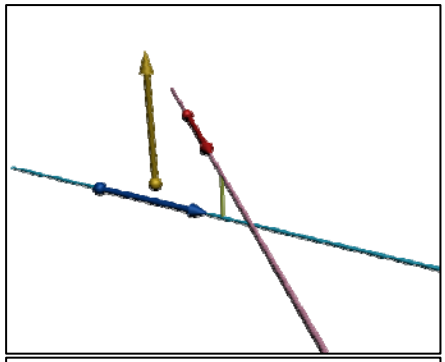
- Caso general:



Los dos rayos están en diferente plano.

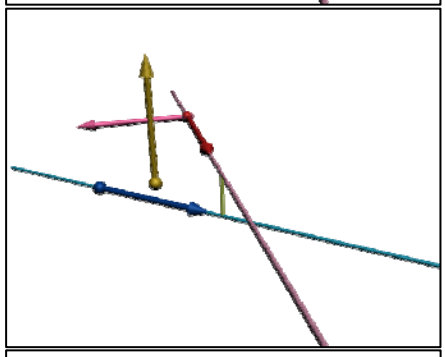
El proceso para calcular la intersección entre dos rayos es:

1.- Realizamos el producto vectorial entre las direcciones de los dos rayos(rayo A y rayo B), a este vector le llamamos cDir, y es perpendicular a ambos. Si el módulo del vector resultante es nulo, quiere decir que dichos rayos son paralelos.
Devolvemos los dos puntos orígenes de los dos rayos, pues los rayos no interseccionan.



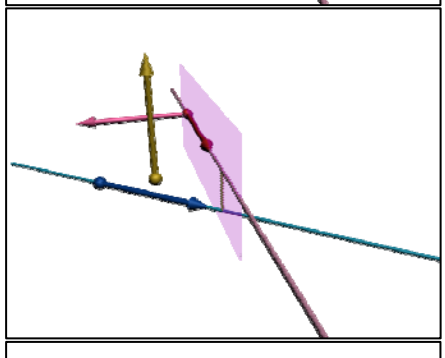
2.- Calculamos el producto vectorial entre la dirección del rayo A y el vector cDir calculado anteriormente.

3.- Con dicho vector normalizado y el punto origen del rayo A calculamos el plano ac.

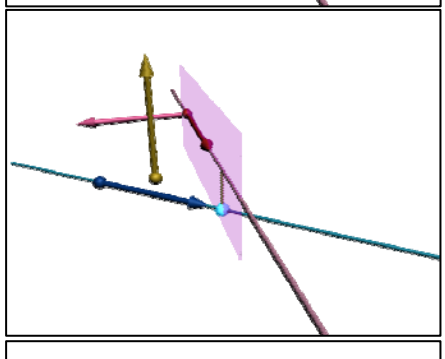


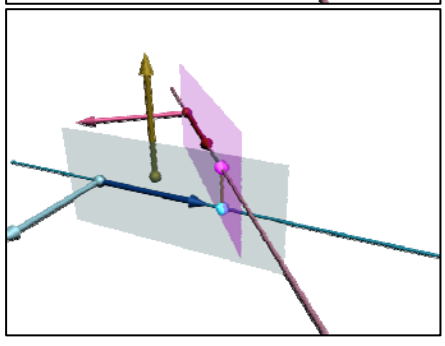
4.- Intersecamos el rayo B inicial con el plano ac. Obteniendo **uno de los puntos solución**.

5.- Análogamente, calculamos el producto vectorial entre la dirección del rayo B y el vector cDir. Con dicho vector normalizado y el punto origen del rayo B calculamos el plano bc.
Análogamente, intersecamos el rayo A inicial con el plano bc. Obteniendo el **otro punto solución**.



3-En el caso en el que la distancia entre los dos puntos solución sea nula, significará que los dos rayos se cruzan, y los dos rayos pertenecen al mismo plano.
Si, por el contrario, la distancia entre los dos puntos es no nula, nos encontramos en el caso general, y los dos rayos no pertenecen al mismo plano.





3.2.5. Cuaterniones y Slerp

Cuaternión

¿Qué es un cuaternión? Matemáticamente, son una extensión del conjunto de números complejos que utilizaremos para albergar componentes de rotación. Su relación con los números complejos, sin embargo, habría que dejarlo para una explicación mas exhaustiva, centrándonos aquí en su utilización para rotaciones en 3D.

Se trata de una entidad matemática compleja pero que, sin embargo, sí puede ser entendida 'a nivel usuario' con una pequeña explicación en esta sección, de manera que seamos capaces de usar un API de cuaterniones y sacar provecho de su principal ventaja: concatenación e interpolación de rotaciones suaves.

El grupo de todos los cuaterniones con el valor absoluto igual a uno, '*cuaterniones de la unidad*' son aquellos que expresan una rotación pura en 3D, al igual que todos los números complejos unitarios son los que expresan una rotación en 2D.

Es una alternativa de representar rotaciones a través de cualquier eje. Podemos pensar en un cuaternión como un vector de 4 dimensiones de la forma:

Q= w+xi+yj+zk

Teniendo:

$$i^2 = j^2 = k^2 = -1$$
$$ij = k,\ ji = -k$$
$$jk = i, kj = -i$$
$$ki = j, ik = -j$$

Donde las componentes x, y, z definen el vector utilizado como eje de modo no normalizado, mientras que la componente w expresa el ángulo de rotación a aplicar. Se escribe también por tanto

Q= [w (x y z)]
Q= [w v]

Sin embargo, w (relacionado con la rotación) y v (con el vector) no están expresados como cabría pensar, a saber, w en radianes y v como un vector normalizado, ¡no puede ser tan fácil !. En lugar de eso, para representar el cuaternión de la rotación de ángulo **θ** sobre el vector **n**, hay que seguir la siguiente fórmula:

Q = [cos(θ/2) sin(θ/2)n]
Q = [cos(θ/2) (sin(θ/2)nx sin(θ/2)ny sin(θ/2)nz)]

Esta representación, lejos de ser un 'capricho' tiene algunas ventajas:

- Todo cuaternión que represente una rotación tiene módulo uno.
- Si el ángulo de rotación es 0 o 360, su seno es 0, lo que anula el vector por completo. Esto es una ventaja, pues así no hay que inventar un eje 'por el que no rotar'.

Otra consecuencia de esta representación es que si negamos una cuaternión (de la manera habitual, negando cada una de sus componentes) y reinterpretamos el resultado como ángulo-vector obtenemos:

Q = -[cos(θ/2) sin(θ/2)n]
Q = [-cos(θ/2) -sin(θ/2)n]
Q = [cos(θ/2 + π) sin(θ/2 + π)n]
Q = [cos((θ + 2 π)/2) sin((θ + 2 π)/2)n]
Q = [cos(θ/2) sin(θ/2)n]

¡Representa la misma rotación! Es decir, para cualquier rotación en el espacio existen exactamente 2 representaciones en forma de cuaternión. Incluso para la rotación identidad: (1 0 0 0) y (-1 0 0 0).

Si negando el cuaternión conseguimos la misma rotación… ¿Cómo conseguimos la rotación inversa?

cos(-θ) = cos(θ)
sin(-θ) = -sin(θ)

Por tanto deberíamos invertir la parte afectada por el seno: el vector **v**.

Nota: A esta operación se le llama conjugado de un cuaternión, para conseguir la inversa deberíamos dividir por el módulo del vector (que se calcula de la manera tradicional), sin embargo, esto no debe preocuparnos mientras tratemos sólo con cuaterniones que representan rotaciones (módulo 1).

La concatenación de rotaciones, al igual que en las matrices, se lleva acabo mediante el producto (¡ordenado!) de cuaterniones, que está definido de una manera similar al producto de números complejos: Basándose en la definición de i, j y k. Se trata sin embargo de una definición relativamente larga que no vamos a abordar.

El resultado final es:

$$\begin{bmatrix} w_1 & V_1 \end{bmatrix} \begin{bmatrix} w_2 & V_2 \end{bmatrix} = \begin{bmatrix} w_1 w_2 - V_1 \cdot V_2 & w_1 V_2 + w_2 V_1 + V_2 \times V_1 \end{bmatrix}$$

Este producto de cuaterniones produce una concatenación de las rotaciones que representan sus operadores completamente suaves, sin problemas de 'gimbal lock'.

Es posible también restas (o dividir) las rotaciones de dos cuaterniones. Sea **a** y **b** dos cuaterniones queremos conocer el cuaternión **d** que lleva desde **a** hasta **b**.

ad=b

Esto se puede conseguir simplemente mediante:

d = a^{-1}b

También están definidas para cuaterniones las funciones **exp** y **log**. Considerando la variable **a = θ/2.** El logaritmo se define como:

$$\log Q = \log[\cos a \quad n\sin a]$$
$$\log Q = [0 \quad an]$$

Y la exponencial como:

$$\exp Q = \exp[0 \quad an]$$
$$\exp Q = [\cos a \quad n\sin a]$$

Estas operaciones, no son completas:

El logaritmo solo acepta cuaterniones modulo uno (rotaciones). Devolviendo cuaterniones sin parte real.

Inversamente, la exponencial solo admite cuaterniones sin parte real, devolviendo cuaterniones modulo uno.

Cumplen además, como es de suponer:

$$e^{\ln Q} = Q$$

Estas operaciones, realmente, tampoco son muy útiles por si mismas, pues no tienen que ver con rotaciones, pero si son herramientas para construir la potencia de cuaterniones, definida como:

$$Q^t = e^{t \ln Q}$$

Que si es realmente interesante para rotaciones, pues permite graduar la 'intensidad' de una rotación: Q^0 es la identidad, Q^1 es la rotación, Q^2 es el doble de la rotación, $Q^{0.5}$ la mitad…

¡Conseguido! Con todas estas herramientas ya podemos definir la interpolación de rotaciones. Pues si la interpolación lineal de números está definida como:

$$slerp(a,b,t) = a + (b-a)t$$

Siendo a, b y t números reales, si redefinimos a y b como cuaterniones unitarios podemos hacer:

$$slerp(a,b,t) = a(a^{-1}b)^t$$

Que podría interpretarse de la misma manera: El primer cuaternión **a** más (por) la diferencia entre b y a ($\mathbf{a^{-1}b}$) graduada entre 0 y 1 por **t** (usando la potencia de cuaterniones).

A esta operación se la llama **Spherical Linear intERPolation**. Dicho tipo de interpolación es útil porque permite sin ninguna pega interpolar entre dos rotaciones.
Es una operación ternaria que acepta tres operandos. Los dos primeros son los cuaterniones entre los cuales queremos realizar la interpolación (orientación inicial y final de la interpolación correspondiente), mientras que el tercero es el tiempo.

Resumiendo, a la hora de escoger una representación para nuestras rotaciones tenemos que tener en cuenta las características e las tres disponibles: Matrices, Ángulos de Euler y cuaterniones.

Características	Matriz	Ángulos eulerianos	Cuaterniones
Aplicar rotación a puntos del espacio	**Posible**	**Imposible (debe convertirse a una matriz)**	**Imposible (debe convertirse a una matriz)**
Concatenación de rotación	**Posible pero lento**	**Imposible, *'gimbal lock'* y otros problemas**	**Posible y rápido**
Interpolación	**Imposible**	**Posible pero con *'gimbal lock'***	**Posible, Slerp permite una suave interpolación**
Interpretación humana	**Difícil**	**Fácil**	**Difícil**
Almacenamiento en memoria	**Nueve números**	**Tres números**	**Cuatro números**
Representación única para una orientación dada	**Si**	**Infinitos ángulos eulerianos pueden dar la misma orientación**	**Dos representaciones para la misma orientación (positiva y negativa)**
Posibilidad de llegar a ser una solución no válida	**Puede ocurrir**	**Cualquier trío de números forman una orientación válida**	**Puede ocurrir**

Como puede verse en la tabla, los cuaterniones son una forma alternativa de representar una rotación a los, más conocidos, Ángulos de Euler y matriz de rotación.

Un cuaternión utiliza menos espacio en memoria para albergar una rotación que su matriz equivalente, pero sí ocupa más espacio que sus ángulos eulerianos. Esto significa que, a los tres grados de libertad que tiene un ángulo en 3D (coincide con los Ángulos de Euler), el cuaternión añade un grado de redundancia (relacionado con la restricción de que sea módulo uno) mientras que la matriz de rotación 3x3 añade ¡6 grados de redundancia!. Mientras más grados de redundancia tenga una representación, más fácil es que, después de varias operaciones, acabe no representando una rotación válida.

Además concatenar cuaterniones exige menos operaciones que las matrices, por lo que es más rápido.

Sin embargo, si ninguna duda su ventaja principal es no sufrir de '*gimbal lock*' al concatenar e interpolar rotaciones, que es el nombre que se le a la anomalía que tienen los Ángulos de Euler cuando trabajan cerca de los polos: Pitch muy cercano a $\pi/2$ o $-\pi/2$.

3.3. Fases de implementación: Algoritmia

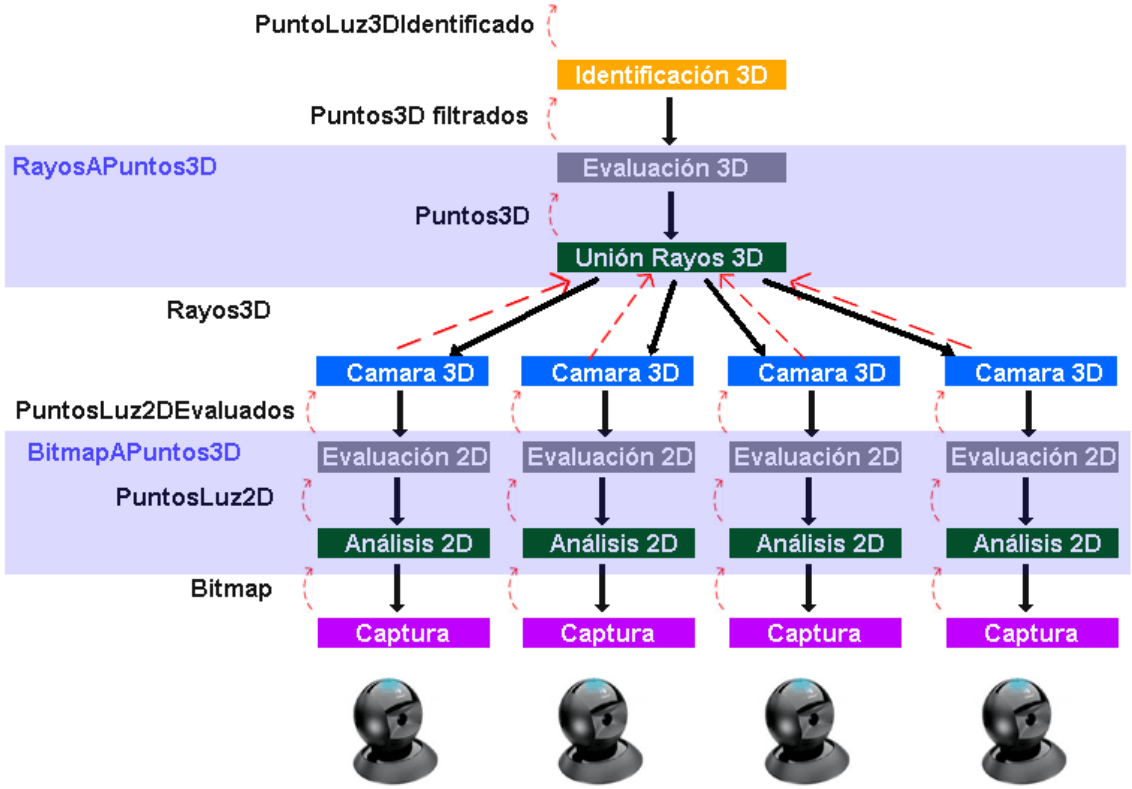
En la mayoría del software que se desarrolla actualmente la algoritmia está perdiendo peso, a favor de las etapas de análisis y diseño, y la reutilización de software. Nuestro proyecto, definitivamente, no sigue esta tendencia.

Al ser un proyecto de investigación en un área relativamente inexplorada, el peso de la algoritmia en nuestro proyecto ha sido muy alto, teniendo una gran importancia la eficiencia y tolerancia a fallos de éstos.

Por la misma razón, no ha habido mucha oportunidad para reutilizar software en nuestro proyecto, teniendo que ser la mayoría de los algoritmos implementados por nosotros o, incluso, inventados por nosotros.

Ejemplos de esto son el caso del algoritmo de reconocimiento 2D, la identificación 3D o, de los que estamos especialmente orgullosos, el algoritmo **pirámide** de reconstrucción 3D, y la autocalibración basada en algoritmos genéticos y cuaterniones.

Aún así, no hemos vacilado en reutilizar software allí donde veíamos la oportunidad, con el ánimo de ser capaces de abarcar más.



Esta ilustración, que utilizamos en el punto '3.1.1.2. Arquitectura Pipeline' para explicar la arquitectura del proyecto y el problema de las dependencias y los eventos puede servirnos ahora como mapa, en el que encajan todos los algoritmos que utilizamos.

De abajo hacia arriba, el proceso comienza en las webcams, que capturan imágenes de las luces, viéndolas cada una desde un ángulo, y se las entregan a DirectShow. A través de código que interopera con DirectShow mediante la tecnología COM (ya un poco obsoleta) somos capaces de generar objetos de la clase Bitmap.

El Análisis 2D consiste en analizar estos Bitmaps capturados y extraer de ahí información muy sintetizada: Una lista de objetos PuntoLuz2D que resumen las características esenciales de cada región brillante encontrada en la imagen.

Inmediatamente después de esto, y de acuerdo con estas características, los PuntoLuz3D son evaluados, y filtrados aquellos que no tienen '*pinta*' de ser luces.

Gracias a que el usuario ha indicado con precisión la posición y orientación de cada cámara, somos capaces de reconstruir RayosLuz3D de cada punto, estos rayos representan las posibles posiciones que puede tener un PuntoLuz2D en el espacio, a la vista de lo observado por esta cámara.

Reuniendo la información aportada por todas las cámaras, los RayosLuz3D, somos capaces de reconstruir la escena 3D y deducir la posición de los puntos. Esta reconstrucción sin embargo no está exenta de problemas.

Para solucionar gran parte de estos problemas, al igual que hicimos en 3D, evaluamos y filtramos los Puntos 3D de acuerdo a sus características.

El paso final consiste en identificar, de la manera más razonable posible, los puntos 3D para poder hacer un seguimiento de ellos.

3.3.1. Captura de imagen

DirectX es la tecnología de Microsoft para el manejo del hardware multimedia de una manera abstracta pero muy eficiente. Se compone de varios módulos:

- **Direct3D**: Permite renderización 3D, incorporada en las tarjetas gráficas modernas.
- **DirectDraw**: Primitivas de dibujado en 2D de alto redimiento.
- **DirectImput**: Dispositivos de entrada, joystick, ratón, mouse…
- **DirectShow:** Captura y reproducción de video
- **…**

DirectX es, hoy por hoy, un estándar en el desarrollo de aplicaciones multimedia y videojuegos. Su utilización habitual es a través de COM, la antigua tecnología de

interoperatibilidad, y los lenguajes más utilizados son C++ y, en menos medida, VisualBasic.

Sin embargo la nueva estrategia .Net de Microsoft rompe, en mayor o menos medida, con todo lo anterior. Esta es la razón de la aparición de Managed DirectX, la versión de DirectX para los lenguajes *manejados* por el CLR: C#, VisualBasic.Net, Managed C++, J#...

Managed DirectX, sin embargo, es una solución provisional ante la aparición de WinFX, el API de programación de Windows Vista, que renueva gran parte de la arquitectura de DirectX, especialmente en el aspecto Gráfico.

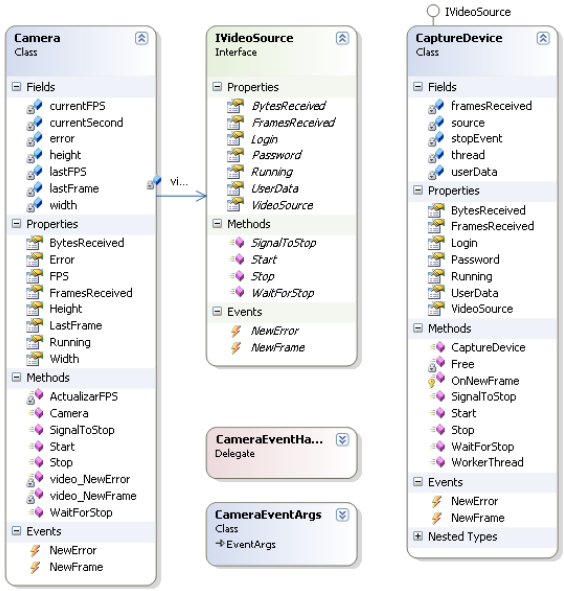
Por este motivo, Microsoft no ha definido una interface *manejada* para la parte de DirectX orientada a video: DirectShow.

Esta inconveniencia lleva a tener que tratar directamente con clases COM, como se haría en C++. Si bien las posibilidades de interoperatibilidad de .Net con COM y con *código nativo* (Win32) son muy buenas, tratar con código COM tiene inconvenientes intrínsecos: La instanciación de objetos, el manejo de GUIDs, las interfaces COM y, sobretodo, la especial atención que hay que prestar a la liberación de recursos hacen del código que interopera con COM mucho más enrevesado que el código .Net *normal*.

Afortunadamente, como comentamos en '2.2.2.2. Componentes Software reutilizados', Andrew Kirillov en su artículo Motion Detection Algorithms de CodeProject
resuelve este problema para implementar, desde C#, su algoritmo de Detección de Movimiento (no confundir con Captura de Movimiento).
(http://www.codeproject.com/cs/media/Motion_Detection.asp)

3.3.1.1. Independencia de la fuente

A pesar de que DirectShow nos abstrae del hardware concreto de captura (modelos de WebCam, o capturadoras de video), Andrew Kirillov llega más lejos, unificando en la interface IVideoSource la captura de video, la reproducción de archivos de video, y la descarga por *streaming* de video desde internet desde diversos formatos.



Aunque en un principio incorporamos la posibilidad de capturar directamente desde archivos de video (en lugar de WebCams) a nuestro proyecto, finalmente no hemos utilizado esta funcionalidad. Aún así el modelo de objetos original se

sigue conservando y la interface IVideoSouce sigue presente.

La clase CaptureDevice implementa la interface IVideoSource para dispositivos de captura de video locales. Es por tanto la que tiene todo el código COM necesario tratar con DirectShow.

Finalmente, la clase Cámara se encarga de manejar, de manera abstracta, cualquier IVideoSource y guardar estadísticas sobre la captura.

3.3.1.2. Configuración de la resolución de las cámaras

Ya avanzado el proyecto, al intentar capturar simultáneamente con cuatro cámaras, nos encontramos con el problema del cuello de botella del USB 1.1: Es imposible trasmitir con nuestras 4 cámaras imágenes a 320x240 a más de 20 frames por segundo.

La única solución a este problema era negociar con la cámara un cambio de resolución, código que debía hacerse a través de DirectShow utilizando COM. Se trata de un código enrevesado, que costó programar y depurar y que dio problemas hasta fases muy avanzadas del proyecto por problemas de liberación de recursos.

Finalmente, pensamos en incorporar esta funcionalidad (cambio de resolución) a la interface IVideoSource, sin embargo el resto de las implementaciones posibles, archivos de video y *streaming* por internet, no disponían de esta capacidad por lo que, en este sentido, el problema era muy similar al que Joel Spolsky, de Joel on Software, explica en su excelente artículo The Law of Leaky Abstractions.
(http://www.joelonsoftware.com/articles/LeakyAbstractions.html)

3.3.2. Reconocimiento de puntos 2D

La primera fase del algoritmo que permite reconocer puntos 3D en el espacio pasa por reconocer puntos 2D en las imágenes entregadas por las fuente: cámara web o un archivo multimedia (AVI) previamente grabado.

Reconocer puntos 2D en cada una de las imágenes conlleva dos problemas fundamentales:

Definir el concepto de punto en 2D: Cada imagen entregada por la fuente consiste en una matriz de colores de resolución fija. Un punto en esa imagen es un grupo de 1 o más píxeles, más o menos contiguos, y de un color más brillante que el resto. Hace falta una manera de traducir esta definición tan imprecisa a algo que el algoritmo pueda procesar. Resolveremos esto evaluando los puntos.

Desarrollar un algoritmo rápido: Existen Mocaps que pueden permitirse procesar a posteriori la información capturada, pero éste no es nuestro caso. Nuestra intención es

crear un HID (Dispositivo de interfaz humana) y por tanto tiene que ser una aplicación de tiempo real. Una primera aproximación al problema revela unos altos requisitos de rendimiento:

De 15 a 60 fps (cuadros por segundo)
De 160 x 120 a 640 x 480 píxeles
De 3 a 6 cámaras simultáneas.

Por tanto:
De 45 a 360 imágenes por segundo simultáneas
De 864.000 a 110.592.000 píxel por segundo simultáneos.

Las conclusiones de esta pequeña estimación son:

- La diferencia en requisitos de rendimiento entre la configuración más exigente y la más modesta es muy alta. Hay que ser cuidadoso jugando con los parámetros de configuración del dispositivo de la misma manera que con cualquier dispositivo de video.
- Sea cual sea la configuración, se trata de un algoritmo que necesita de una gran velocidad, pues cada operación de más que se haga en un píxel puede repetirse más de 110 millones de veces por segundo.

Para resolver este problema aplicaremos una versión modificada del algoritmo QuickFill.

3.3.2.1. Algoritmo QuickFill

QuickFill es una de las varias implementaciones posibles de un algoritmo de Flood Fill. La idea intuitiva de un algoritmo de Flood Fill es conocida por cualquier persona que haya usado un programa de retoque fotográfico como Adobe Photoshop: La varita mágica, que permite seleccionar una región continua de un color similar.

La implementación básica de éste tipo de algoritmos consiste en llamadas recursivas hacia 4 (u 8) direcciones, viendo cada vez si el color del nuevo píxel mantiene una distancia (d) con el color del 'píxel original' menor que la tolerancia (t). En caso de ser así se marca el píxel como visitado y se incluye en el conjunto solución.

QuickFill mejora ésta implementación en dos sentidos, uno más teórico y otro más práctico (a bajo nivel):

- La implementación intuitiva, al hacer llamadas recursivas de una manera desordenada, sufre de una gran cantidad de **colisiones** en píxeles ya visitados (intentos de visitar un píxel más de una vez). Además tiende a ser una **búsqueda en profundidad** haciendo un mal uso de la Pila de Llamadas (llegando a 1 millón de llamadas recursivas).

- Otra consecuencia de la implementación intuitiva es que **no aprovecha la estructura interna** en la que están almacenados los **pixels de un *bitmap*** en memoria: De izquierda a derecha y de arriba abajo. Cada una de estas líneas (*scanline*) está, además, alineada con el tamaño de palabra de la memoria de la arquitectura (4 bytes). A este desplazamiento de le llama *stride*.

Debido a esta disposición de los píxel, un recorrido de la imagen eficiente ha de potenciar los movimientos horizontales a los verticales, pues son mucho más eficientes tanto al calcular la nueva posición como en accesos a caché.

Basándonos en el código en C++ de John R. Shaw en su artículo de CodeProject http://www.codeproject.com/gdi/QuickFill.asp hemos variado la implementación de QuickFill, éstas son sus características:

- No comienza la exploración en un punto dado (varita mágica), sino que explora toda la imagen en busca de regiones con distancia a un color dado menor que la tolerancia. Cada vez que encuentra una añade un nuevo PuntoLuz con información sobre la región.
- Hace llamadas recursivas, pero solo verticales, al igual que QuickFill.
- Evita las colisiones más simples y comunes.
- Está implementado en C# utilizando código *unsafe* que permite la utilización de punteros para mejorar la eficiencia.

Nuestro algoritmo, básicamente, recorre mediante código unsafe toda la imagen mientras los colores de ésta no sean 'buenos', es decir, mientras considere que estamos en el fondo. Recorrer la imagen supone ir moviéndose por cada uno de los píxel, y por cada una de las casillas del array de bool que determina si un píxel está o no visitado. Se necesitan por tanto 3 contadores distintos para acceder a estos valores:

- Coordenadas **x** e **y** de la imagen para limitar los bucles
- Puntero al píxel actual: **pPixel** = posBitmap + y*tamScanLineEnPixels+ x*tamPixel
- Posición en el *array* de bool de posiciones visitadas (ba): **posXY** = y*width + x

Para evitar hacer tantas multiplicaciones por píxel, estos contadores se mantienen sincronizados.

```
public List<PuntoLuz> BuscarPuntos()
{
    Point size = PixelSize;
    List<PuntoLuz> lpl = new List<PuntoLuz>();
    LockBitmap();
    int posXY=0;
    for (int y = 0; y < size.Y; y++)
    {
        PixelData* pPixel = PixelAt(0, y);
        for (int x = 0; x < size.X; x++, posXY++, pPixel++)
        {
            if (!ba[posXY] && BuenColor(pPixel))
            {
                int nx;
                PuntoLuz pl = SuperFill(x, y, out nx, pPixel, posXY);
                lpl.Add(pl);
                int inc = nx - x - 1;
                x += inc;
                posXY += inc;
                pPixel += inc;
```

```
            }
        }
    }
    UnlockBitmap();
    return lpl;
}
```

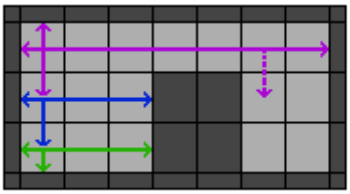
Una vez que se encuentra en la imagen un punto no visitado y con un 'buen color' se llama al algoritmo SuperFill que explora la región y devuelve un PuntoLuz con información sobre ésta. SuperFill es, básicamente, una entrada fácil al algoritmo recursivo Fill que hacíe lo siguiente:

- Comienza en el punto (x, y), y explora hacia la izquierda (xp) y hacia la derecha (xn) hasta que se encuentren píxel que no sean 'buen color' o hayan sido visitados, o se llegue al fin de la imagen. Cada nuevo píxel visitado se incluye en la información de la región (PuntoLuz).
- Después se hacen llamadas recursivas hacia arriba (y-1) y hacia abajo (y+1), tantas como sean necesarias para explorar todo el intervalo desde [xp,xn] (cada llamada recursiva informa de la coordenadas x hasta la que ha llegado).

De ésta manera se reduce considerablemente el número de llamadas recursivas (proporcionales a la altura de la región, y no al área), y, sumado a que en nuestro caso las regiones son de a lo sumo 10 x 10 píxel (puntos de luz), un problema de desbordamiento de pila se hace prácticamente imposible.

Tiene sin embargo aún un pequeño problema: Cuando se hace una llamada recursiva, hacia arriba por ejemplo, tarde o temprano ésta va a hacer una llamada recursiva hacia abajo. Ésta llamada es evitable si cada una 'sabe' en que dirección va.

Más en detalle, una llamada que va en una dirección, puede evitar hacer una llamada en la dirección contraria siempre y cuando ésta no se encuentre en los píxeles que fueron visitados por la llamada anterior, veamos un ejemplo:



La primera llamada recursiva explora horizontalmente, después intenta hacer llamadas recursivas hacia arriba para los 7 píxel que 'abarca', sin éxito. Finalmente crea una segunda llamada recursiva hacia abajo. Ésta, después de explorar horizontalmente, **no intenta hacer una llamada recursiva hacia arriba** pues sabe que ella va hacia abajo y solo debe hacer llamadas hacia arriba si amplia la región horizontal explorada por su padre (no es el caso). Así que hace una llamada recursiva hacia abajo directamente. El proceso después se retoma por la primera llamada recursiva.

```
private int Fill(ref PuntoLuz pl, int x, int xf, int y,  Direct dir, PixelData* p,int
posXY )
{
    pl.AnadirPos(x, y, p);
    if (dir == Direct.Arriba) pl.posMin.Y = Math.Min(pl.posMin.Y, y);
    else if (dir == Direct.Abajo) pl.posMax.Y = Math.Max(pl.posMax.Y, y+1);
```

```
    ba[posXY] = true;

    int xp = x - 1;
    PixelData* pp = p - 1;
    int posXYp = posXY - 1;

    // recorrido del punto a la izquierda
    for (; xp >= 0 && !ba[posXYp] && BuenColor(pp); xp--, pp--, posXYp--)
    {
        pl.AnadirPos(xp, y, pp);
        pl.posMin.X = Math.Min(pl.posMin.X, xp);
        ba[posXYp] = true;
    }
    xp++;
    posXYp++;
    pp++;


    int xn = x + 1;
    PixelData* pn = p + 1;
    int posXYn = posXY + 1;

    // recorrido del punto a la derecha hasta el tope xf no es necesario tenerlo en
cuenta
    for (; xn < rWidth && !ba[posXYn] && BuenColor(pn); xn++, pn++, posXYn++)
    {
        pl.AnadirPos(xn, y, pn);
        pl.posMax.X = Math.Max(pl.posMax.X, xn+1);
        ba[posXYn] = true;
    }


    if (dir == Direct.Ninguna)
    {
        #region Arriba Normal
        ...
        #endregion

        #region Abajo Normal
        ...
        #endregion
    }


    else if (dir == Direct.Arriba)
    {
        #region Arriba Normal
        if (y > 0)
        {
            PixelData* pi = (PixelData*)((byte*)pp - stride);
            int posXYi = posXYp - rWidth;
            int xi = xp;
            for (; xi < xn; xi++, pi++, posXYi++)
            {
                if (!ba[posXYi] && BuenColor(pi))
                {
                    int inc=Fill(ref pl, xi, xn, y - 1, Direct.Arriba, pi, posXYi)-xi-1;
                    xi += inc;
                    pi += inc;
                    posXYi += inc;
                }
            }
        }
        #endregion

        #region Abajo Alerones
        if (y < rHeight - 1)
        {
            PixelData* pi = (PixelData*)((byte*)pp + stride);
            int posXYi = posXYp + rWidth;
            int xi = xp;
            for (; xi < x; xi++, pi++, posXYi++)
            {
                if (!ba[posXYi] && BuenColor(pi))
                {
                    int inc =Fill(ref pl, xi, xn, y + 1, Direct.Abajo, pi, posXYi)-xi-1;
                    xi += inc;
```

```
                    pi += inc;
                    posXYi += inc;
                }
            }
            {
                int inc = xf - x;
                xi += inc;
                pi += inc;
                posXYi += inc;
            }
            for (; xi < xn; xi++, pi++, posXYi++)
            {
                if (!ba[posXYi] && BuenColor(pi))
                {
                    int inc =Fill(ref pl, xi, xn, y + 1, Direct.Abajo, pi, posXYi)-xi-1;
                    xi += inc;
                    pi += inc;
                    posXYi += inc;
                }
            }

        }
        #endregion
    }

    else if (dir == Direct.Abajo)
    {
        #region Arriba Alerones
        ...
        #endregion

        #region Abajo Normal
        ...
        #endregion
    }

    return xn;

}
```

Finalmente, cuando decidimos detectar el color de las luces, modificamos el algoritmo.

Las luces son reconocidas como regiones blancas con un pequeño reborde del color propio de la luz. Hay que prestar por tanto especial cuidado a como se trata el color blanco en la imagen.

Existen una *variable global* colorNeutro que contiene la tonalidad de la zona brillante de la luz, blanco en casi todos los casos. Existe además un *array* con los colores objetivos que se han de buscar (los colores de las luces). Éstas variables se cargan de un archivo de configuración.

Un punto de luz tiene, ahora, un color asociado, que puede ser el color neutro, o un color objetivo. Si todos los píxel son de colorNeutro, entonces el punto es de colorNeutro. Si se encuentra algún píxel de un color objetivo, entonces la región pasa a ser de ese color objetivo, e incompatible con cualquier otro color objetivo, pero no con el color neutro (blanco).

Con esta técnica se consigue identificar como puntos de luz rojos, por ejemplo, aquellos que sean mayoritariamente blancos pero con algún píxel rojo.

Para hacer esto, sin embargo, es necesario capaz de identificar cualquier color RGB que pueda aparecer en una imagen con su color objetivo más cercano. La manera sencilla

de hacer esto es recorrer la lista de colores objetivos y calcular la distancia a cada uno de ellos quedándonos con el más cercano.

Esto supone para cada píxel de la imagen un costoso for con operaciones aritméticas dentro que podría tirar abajo el rendimiento del análisis 2D, la etapa más problemática del proyecto en cuestiones de rendimiento.

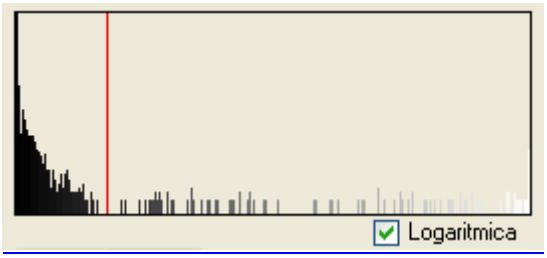
Para evitar ésta sobrecarga existe la clase CuboColores, que precalcula un cubo RGB de manera que, dado un color, conocer con que color objetivo está asociado (o colorNeutro) es de coste constante.

3.3.2.2. Histograma

Un punto fundamental del análisis de las imágenes producidas por las cámaras es detectar si un píxel es suficientemente brillante. Éste filtro se aplica antes de intentar identificar el color del píxel.

Determinar a partir de que valor considerar un píxel brillante no es un problema sencillo, pues depende por completo de las condiciones de luminosidad de la estancia donde se estén capturando las imágenes.

Para simplificar éste proceso generamos un histograma, que permite ver de una manera global la distribución estadística de la luz en una imagen. Éste histograma se recalcula cada ciertos *frames*.



3.3.2.3. Evaluación de puntos

Hasta el punto actual, hemos conseguido analizar sistemáticamente todos los pixels de una imagen capturada por las webcams, *recolectando* un a lista de PuntoLuz2D que dan información sobre la regiones reconocidas. Éste proceso está optimizado para ser lo más rápido posible, a costa incluso de llamar PuntoLuz2D a regiones que, realmente, no lo son.

Se hace necesaria, por tanto, una segunda etapa que filtre los PuntosLuz2D que representen a reflejos, brillos, ventanas, etc… Aislar éste filtrado a un proceso posterior al análisis de la imagen mejora el rendimiento, pues trabaja con PuntosLuz2D (del orden de decenas) en lugar de con píxel (del orden de millones).

No existen, sin embargo, un conjunto de características que determinen sin error si un PuntoLuz2D es suficientemente bueno, por lo que valoramos la calidad de un punto asignándole una nota en función del su parecido a las de un hipotético *punto de luz ideal.*

Las características que valoramos en un PuntoLuz2D son tres:

- **Tamaño**: Un *punto de luz ideal* tiene un tamaño óptimo que se puede determinar empíricamente (configurable).
- **AspectRatio**: La proporción entre la anchura y la altura ha de ser lo más parecido a 1, lo que indica que se trata de un punto relativamente cuadrado, discriminando reflejos que son a menudo más alargados.
- **Densidad de puntos**: Además, un buen punto debe ser circular. Determinar ésta característica de manera precisa es costoso, pero existe una manera rápida y empíricamente válida. Consiste en comparar la proporción del número de píxeles luminosos entre el área del cuadrado que los envuelve con la del área de un círculo y el cuadrado que lo circunscribe ($\pi/4$).

Hemos seguido una aproximación más cercana a la lógica difusa que a la lógica clásica, lo que creemos que es permite evaluar de una manera más *justa* la calidad de los puntos de luz.

Detalles de implementación de la función de evaluación:

- **Distancia Geométrica**: Hemos definido ésta función como:

```
private float DistanciaGeometrica(float a, float b)
{
    if (a < b)
        return a / b;
    return b / a;
}
```

  Sus propiedades principales son que devuelve 1 cuando a = b y éste numero va decreciendo a medida que a y b se alejan. Es simétrica por tanto DistanciaGeometrica(a,b) = DistanciaGeometrica(b,a). En principio su definición solo es útil para números positivos.
  Utilizando un parámetro como el deseado y el otro como el estudiado, devuelve el índice de similitud entre uno y otro.

- **Producto de las notas parciales:** Multiplicando, en lugar de haciendo la media, la valoración de cada una de las características parciales se fuerza a que cada característica deba ser 'suficientemente buena'. Unas valoraciones más equilibradas son una buena idea en este caso.

3.3.3. Reconstrucción de rayos 3D

Durante etapas anteriores hemos estado tratando los puntos de luz como regiones en un Bitmap bidimensional, las hemos reconocido, evaluado y seleccionado. El proyecto sin embargo pretende reconstruir las posiciones en 3D de estos puntos de luz gracias a que son vistos desde múltiples ángulos. Convertir estos puntos en 2D a rayos en 3D, a base de deshacer la proyección, es el primer paso para reconstruir este espacio tridimensional.

Para poder dar éste paso no se necesita información extra de ningún tipo de sensor, ni un tipo especial de cámara. Tan solo es necesario conocer con exactitud la posición y orientación de la cámara.

3.3.3.1. Orientación de las cámaras

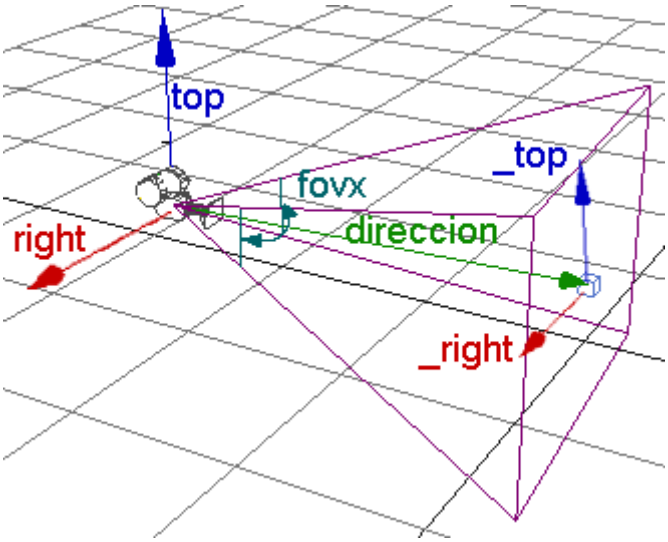
Los datos necesarios para saber como está colocada la cámara son, un Vector3D para la posición y dos vectores para la orientación: dirección y top (top es necesario porque hay que tener en cuenta que la cámara no tiene por qué estar necesariamente bocarriba). Además se puede calcular el vector right simplemente como el producto vectorial de top y dirección.

Además, una cámara tiene dos constantes que definen su proyección: el aspectRatio, que es el cociente entre la anchura y la altura de la imagen, y el fieldOfView, que define la apertura de la pirámide de visión, en nuestro caso la apertura sobre el eje x (anchura).

Con estos datos ya es posible generar un rayo para cada punto de luz encontrado, entendiendo por un rayo un objeto matemático que tiene un origen, y una dirección hacia la que se extiende. El origen de cada rayo es, simplemente, el origen de la cámara, mientras que para calcular la dirección se utilizan los vectores precalculados _right y _top, de manera que se puede calcular un rayo en dos sencillos pasos.

- Transformar la posición del punto de coordenadas de la pantalla ([0…319], [0...239]) a coordenadas homogéneas ([-1...1], [1...-1]).
- La dirección del rayo es: [dirección de la cámara] + x * _right + y * _top.

_right y _top son por tanto vectores libres paralelos a right y top pero de distinta longitud, de manera que al situarlos en posición + dirección, son exactamente la mitad de grandes que el plano de Frustum que se encuentra a distancia 1 de la cámara. Se ve bien en la siguiente imagen:



Estos datos son pedidos al usuario y se pueden guardar en un archivo de configuración. Sin embargo, creer que el usuario va a ser capaz de escribir las coordenadas de la posición, y que lo va a hacer con la precisión necesaria, es creer demasiado. Es por ello que, además de una aplicación que permite configurar de manera visual la manera en la que están dispuestas las cámaras, se hace necesario que las cámaras sean capaces de calibrar automáticamente la colocación que el usuario les ha asignado, especialmente su orientación.

3.3.3.2. Algoritmos genéticos de auto configuración

Como hemos visto, para que los PuntosLuz3D puedan ser transformados en Rayos es necesario dar a Camara3D información fiable de su colocación en la escena. Cuantos más fiables sean estos datos, mas precisión tendrán los rayos generados por Camara3D, lo que repercutirá en la calidad de los PuntosLuz3D que se generarán en pasos posteriores.

Para un usuario, sin embargo, es muy difícil conocer las coordenadas donde está situada su WebCam o, lo que es más importante, definir con precisión su orientación. Es necesario que la cámara, basándose en lo que ve, sea capaz de refinar los datos aportados por el usuario.

El problema consiste en averiguar una función que dados:

- Coordenadas en 3D de unos puntos fijos en el mundo.
- Coordenadas en 2D de donde han sido visto esos puntos en la imagen capturada.

Sea capaz de deducir los siguientes datos de la cámara:

- Posición
- Dirección

- Top

Resolver éste problema por técnicas analíticas (se trata de un sistema no lineal) o geométricas (extensiones del arco capaz a 3D por ejemplo) es una tarea complejísima, que podría constituir un proyecto en sí mismo.

Lo que si es factible es evaluar la calidad de una solución dada: Construimos una Camara3D con los datos de la solución posible (posición, dirección y top) y proyectamos los puntos en 3D en ésta para, ya en el plano 2D, medir la distancia entre estos puntos proyectados y los puntos reales.

Teniendo a una mano una compleja función no lineal, y en la otra una manera de evaluar la calidad de una posible solución, la solución evidente pasa por ¡utilizar algoritmos genéticos para resolverlo !. Tan sólo haría falta ser capaz de definir la operación de mutación y reproducción entre dos individuos (Posición y Orientación de una Cámara). Es necesario, además, conseguir representaciones y operaciones eficientes en memoria y procesador, pues la calidad de un algoritmo genético depende del número de generaciones e individuos que sea capaz de simular.

Mutar una solución de manera razonable es factible:

- Variar la posición es muy simple, tan solo es necesario incrementar conjunta o independientemente cada una de las coordenadas (x, y ó z).
- Variar la orientación, sin embargo, es más complicado. Dirección y top han de permanecer perpendiculares en el proceso, variando de manera solidaria. La solución evidente es generar right (como el producto vectorial de top y dirección) y con los tres vectores generar una matriz a la que se post/pre multiplica por las matrices estándar de rotación por cada uno de los ejes (x, y ó z). Después extraer los vectores top y dirección de la nueva matriz modificada. Se trata de una solución costosa pero posible.

Es con la reproducción cuando se encuentran una gran traba: La mezcla entre las posiciones de dos soluciones se puede definir como la combinación lineal de éstas, pero no ocurre así con la mezcla de sus orientaciones, imposible de definir sin salirse de una representación vectorial (dirección y top) o matricial de las rotaciones.

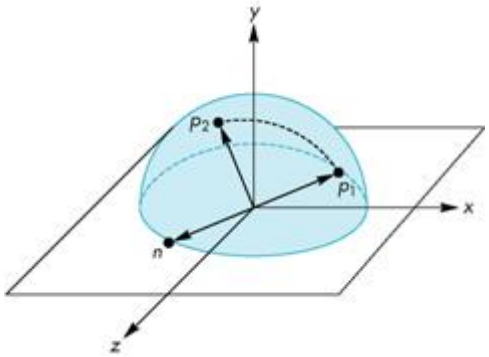
Para hacer posible la interpolación de orientaciones hace falta representarlas de una forma distinta.

Los Ángulos de Euler (yaw, pitch, y roll) son la manera más corta y humanamente comprensible de representar una rotación, pero la definición de interpolación sufre de Gimbal Lock (comportamiento anómalo en los polos)

Existe sin embargo otra manera de representar las rotaciones en 3D, los cuaterniones, que son eficientes en memoria y procesador, y en los que la interpolación ¡está definida de una manera suave incluso en los polos!

Los cuaterniones son a las rotaciones en 3D lo que los números complejos a las rotaciones en 2D. De la misma manera, solo los cuateriones de módulo uno representan rotaciones 'puras'. La idea es que **toda rotación puede ser escrita como un vector y un ángulo de rotación,** pero representada además de cierta manera que, si el ángulo de rotación es cero, no es necesario especificar un ángulo de rotación neutro o 'mentiroso'.

Tan sólo un cuaternión es necesario para almacenar la orientación de la cámara (4 floats), y las operaciones como concatenación de rotaciones o SLERP (spherical linear interpolation) son eficientes.



Redefinir la mutación de la orientación de la cámara con cuaterniones no podría ser más sencillo: Simples pre/post multiplicaciones por los cuaterniones de rotación por cada uno de los ejes (x, y, ó z).

Finalmente, creamos una clase ligera, PosicionCamara, que contiene sólo los datos imprescindibles de un individuo en el algoritmo genético: el Vector3 posición y el Quaternion orientación, para poder maximizar el numero de individuos simultáneos.

Finalmente las características del mejor individuo PosicionCamara son trasladadas a la Camara3D que está siendo autocalibrada. Este es el motivo por el que una Camara3D tiene, además de top y dirección, un cuaternión redundante que define su orientación.

3.3.4. Reconstrucción de puntos 3D

Llegados a éste punto, hemos transformado los PuntoLuz2D encontrados de la imagen (al menos aquellos que parecían mas aptos) en RayosLuz3D, basándonos en la posición y orientación de las cámaras.

El paso siguiente es reconstruir la información de las coordenadas reales de los puntos en 3D, que coincide con los posibles cruces de rayos de distintas cámaras.

Esta tarea es sin duda la etapa más complicada del proceso, estos son sus requisitos.

3.3.4.1. Requisitos

Obviando todos los posibles inconvenientes, el problema podría resolverse para dos cámaras comprobando todos los rayos de la primera con los de la segunda, y si estos rayos se cruzan, generamos puntos en 3D en esa posición.

Sin embargo hay muchos inconvenientes posibles:

- Realmente la distancia entre dos rayos nunca será 0.
- Puede haber más de dos cámaras.
- Cada cámara procesas sus puntos y genera los rayos en **su propio hilo**, por lo que es necesario sincronizar las cámaras.
- **Un punto puede desaparecer** para algunas cámaras y no para otras, o incluso para todas a la vez.
- **Pueden aparecer puntos** falsos, intersecciones de rayos que, realmente, provienen de puntos distintos.
- **Dos puntos pueden fusionarse**, formando un punto mayor y ligeramente deformado, desde el punto de vista de una cámara. El análisis 2D no detectará estos, incluso podría despreciar el punto de luz por ser demasiado grande.
- Dependiendo de la calidad de las cámaras y las luces, **el color de un punto puede no ser muy estable**.
- Y, aunque será tratado en el punto siguiente, es necesario **evaluar e identificar los puntos**.

De estos problemas, algunos pueden ser bien resueltos y otros no tanto. Estos son los requisitos de nuestro proyecto para esta fase del proceso:

Sincronización:

Es necesario que los hilos de las cámaras se sincronicen a cada frame, minimizando la distancia temporal entre el momento de captura de imagen de cada frame (**desfase de frames**) y que el tiempo de procesador no se desequilibre (pues los fps totales son, siempre, los de la cámara más lenta).

Tolerancia a errores:

El algoritmo de reconstrucción ha de ser tolerante a que los rayos no se crucen **exactamente**. Se trata de un error tan extremadamente corriente (ocurre siempre) que ha de ser considerado en el funcionamiento normal de la aplicación.

Además, ha de definir que ocurre cuando una cámara deja de ver puntos, un punto desaparece para dos cámaras, dos puntos se fusionan a la vista de una o varias cámaras, etc.…

Simetría:

Es necesario que todas las cámaras sean igualmente influyentes en el proceso, pues no se puede confiar que una, o un grupo de ellas, vayan a ver siempre los puntos.

Evaluación e Identificación:

Al igual que para los PuntoLuz2D, no existe un criterio para dividir los PuntosLuz3D reales de los errores, pero se pueden evaluar estos puntos a partir de una función y determinar una *nota de corte*.

Igualmente, es necesario preservar la identidad de un PuntoLuz3D de una captura a otra (realmente, de una unión de capturas de cada cámara a otra).

Estos dos procesos, sin embargo, son suficientemente complicados por si mismo como para merecer un apartado: 3.3.5. Evaluación de puntos 3D y 3.3.6. Identificación de puntos 3D.

3.3.4.2. Sincronización

En concurrencia, se llama sincronización a la comunicación de eventos temporales, bloqueos, y señales que permite colaborar a distintos hilos de ejecución.

En muchas aplicaciones la concurrencia es una opción, que permite hacer un programa con una interface de usuario más *responsive* o, en algunos casos, desarrollar algoritmos más eficientes.

En otras aplicaciones, sin embargo, es prácticamente inevitable la concurrencia, como es el caso de un servidor web o de aplicaciones. Estas aplicaciones, sin embargo, esquivan la mayoría de los problemas de concurrencia y mejoran la escalabilidad haciendo que las consultas sean *stateless*.

En nuestro proyecto, al igual que los servidores web, la concurrencia es insalvable, pues ¡Cada cámara captura imágenes en su propio hilo!. Además, si se integra con aplicaciones de terceros, el bucle de esta aplicación (bucle de mensajes de windows o el bucle principal en un juego) poco tiene que ver con nuestro bucle de captura de puntos, lo que sugiere que debe usarse en otro hilo.

Sin embargo cada hilo no goza del nivel de aislamiento que tienen los hilos de un servidor web. En nuestro caso, cada ciclo del hilo de cada cámara ha de aportar la solución parcial, su lista de rayos, que se reunirá para formar la solución total: los puntos 3D, posteriormente evaluados e identificados.

Esta problemática hace de nuestro proyecto un proyecto *rico* en problemas de concurrencia, haciendo uso de multitud de hilos y de primitivas de sincronización avanzadas, como la barrera.

Antes de adentrarnos en analizar y resolver los problemas de sincronización del proyecto, estudiaremos brevemente las posibilidades de sincronización que .Net nos ofrece al respecto:

3.3.4.2.1. Sincronización en .Net

El concepto de hilo es en .Net, al igual que en Java, un *first citizen*, a diferencia de C++ en el que los hilos están incluidos como librerías externas, a veces estandarizadas y a veces dependientes del sistema operativo.

En este apartado no pretendemos, ni por asomo, dar una explicación detallada de todos los mecanismos de hilos y sincronización que permite .Net (puede encontrarse una explicación detallada en http://www.albahari.com/threading/). Tan solo explicaremos los mecanismos de sincronización más comunes, en concreto los que hemos usado en nuestra solución.

Monitor: La clase estática Monitor permite el control de regiones críticas, utiliza simples **object** como *llaves* que definen una región crítica. De esta manera se da mucho *contenido semántico* a las regiones críticas pudiendo hacer algo como:

```
List<string> names = ...;

   try {
     Monitor.Enter (names);
     names.Add("John");
   }
   finally { Monitor.Exit (names); }
```

Lock: Puesto que Enter y Exit son, con mucho, los método más usados de la clase Monitor, C# implementa un constructo que simplifica la gestión de regiones críticas. Su nombre es lock y su sintaxis es similar a la de un if o un while. El Código anterior podría ser reemplazado simplemente por:

```
lock(names)
{
   names.Add("John");
}
```

EventWaitHandle: Si, en lugar de regiones críticas lo que se necesita es ser capaces de hacer a un hilo esperar hasta que reciba una señal de otro hilo (sin el consumo de CPU de la espera activa), la mejor solución es utilizar objetos de tipo EventWaitHandle.

Su utilizacion se basa en el uso del los métodos WaitOne() para hacer esperar el hilo actual, y Set() para señalizar a los hilos que esperan que pueden continuar. Existen dos alternativas (herencias) de esta clase: ManualResetEvent y AutoResetEvent:

ManualResetEvent actúa como una puerta: Se abre llamando a Set(), y se cierra llamando a Reset(). Mientras esté abierta las llamadas a WaitOne() *no hacen nada*, si está cerrado las llamadas serán bloqueantes (pero sin consumo de CPU) hasta que Set() sea llamado (por otro hilo).

AutoResetEvent, opuestamente, actúa como un torno de metro: Se abre llamando a Set(), y en cuanto un hilo llame a WaitOne(), se cerrará automáticamente, sin necesidad de llamar a Reset().

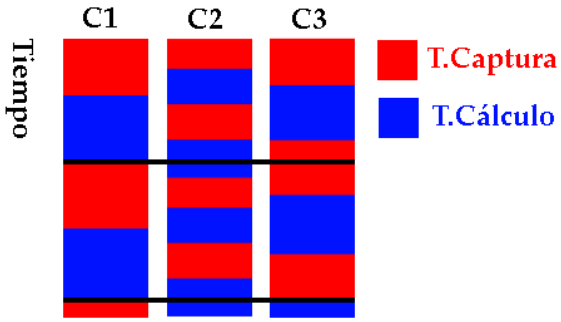
Es importante no confundir conceptos debido al nombre de estas primitivas de señalización: Los **eventos** de .Net (explicados en 3.4.1.1. Delegados y Eventos) no tienen nada que ver con los hilos o la sincronización.

Wait and Pulse: La clase Monitor, además de Enter y Exit, tiene los métodos estáticos Wait, Pulse y PulseAll. Se trata primitivas de señalización de más bajo nivel que los eventos de señalización anteriores y al estar basadas en Monitor sólo permiten sincronización entre hilos (no procesos) lo que las hace también más rápidas.

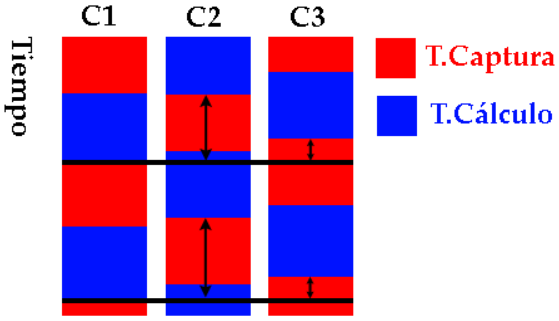
3.3.4.2.2. Problema de Sincronización

Como vimos en los requisitos generales de la reconstrucción 3D, los requisitos son dos:

- Equilibrar los fps de cada cámara, pues los fps totales son los fps de la cámara más lenta. Si una cámara consigue más fps a costa de tiempo de CPU que le quita a otra, estará degradando el rendimiento. En la ilustración se muestra el comportamiento que tendría con una cámara lenta (C1), una rápida (C2) y otra de velocidad media (C3). Las marcas negras indican los puntos de Join, momentos en los que se tiene información nueva de cada cámara y se puede reconstruir una escena 3D.



- Además, aunque todas las cámaras tengan unos fps similares, es interesante que el disparo de cada una de las cámaras se produzca lo más simultáneamente posible, pues mejora la calidad de la reconstrucción 3D. De otra manera, la información de una cámara atrasada o adelantada al resto podría aportar datos que empeorarían la representación 3D si los puntos están en movimiento.



Por tanto, la mejor situación es que todas las cámaras capturen su imagen en el menor intervalo de tiempo, por lo que tendrán que procesarlas simultáneamente.

Desde el punto de vista de rendimiento esta no es la mejor opción, pues la trasmisión de la imagen (tiempo de E/S) de una cámara podría solaparse con el tiempo de procesamiento (tiempo de CPU) de otra.

Sin embargo, en la práctica el tiempo de procesamiento es muy corto en comparación con el de captura y transmisión, y lo realmente importante es la calidad de los datos.

Para conseguir este objetivo (minimizar la distancia entre los tiempos de captura) es necesario contar con una primitiva de sincronización: La Barrera.

La Barrera:

Una barrera es un objeto que contiene tan sólo dos métodos, el constructor Barrera (int threads) y el método Wait(). Su objetivo es hacer que varios threads se esperen en la barrera, y una vez hayan llegado todos, continúen juntos.

El constructor determina de cuantos threads será la barrera (N), mientras que el método Wait() es bloqueante para todos los threads excepto el último, que libera a todos los demás y les permite continuar.

Nuestra implementación inicial de la barrera era costosa en recursos y en tiempo de ejecución, utilizaba 3*N ManualResetEvent.

Gracias a, Joseph Albahari, autor de la web anteriormente comentada, que nos realizó específicamente una mejor Barrera utilizando Wait and Pulse, ahora tenemos una implementación mucho más rápida, y eficiente en uso de recursos.
(http://www.albahari.com/threading/)

Ahora que tenemos una clase barrera que nos permite sincronizar un número determinado de hilos en un punto, haciendo que se esperen unos a otros, la pregunta es ¿Dónde situamos la llamada a Wait() en el proceso?.

Existen dos alternativas principales:

- **Después del T.Cálculo y antes del T.Captura del frame siguiente:** Si seguimos esta alternativa conseguiremos que todos los threads partan en igualdad de condiciones al comenzar la captura de frames, sin embargo nada evita que cualquiera de ellos, después de capturar, continúe haciendo el procesamiento 2D. Hemos visto que esto no es deseable pues atrasaría a las otras cámaras, que aú no han capturado su imagen.

- **Entre el T.Captura y el T.Procesado:** De esta manera se garantiza que se esperará a que todas hayan capturado su imagen para comenzar el procesado. Esta espera sin embargo puede ser potencialmente alta pues puede que alguno de esos threads no haya procesado aún los puntos de la imagen anterior.

La solución a este compromiso es, simplemente, ¡utilizar dos barreras!. Una justo antes de comenzar la captura de imágenes, y una justo después. De esta manera se minimiza el tiempo que puede pasar entre la adquisición de imágenes de cada cámara.

Sin embargo, la barrera situada justo al principio de la adquisición de las cámaras y, por tanto, al final del procesamiento de puntos ha de tener un comportamiento especial. Es necesario que comunique, mediante un evento, que se han producido un Join, o lo que es lo mismo, que tenemos información actualizada de todas las cámaras (rayos 3D nuevos).

A partir de este evento se lanzará el proceso de reconstrucción 3D, evaluación, e identificación… pero, ¿que hilo debería resolver este trabajo? Si lo hiciera un hilo de las cámara, además de ser una solución muy asimétrica, desequilibraríamos la carga hacia de esa cámara empeorando el rendimiento y la calidad de los resultados. El hilo principal tampoco debe resolverlo, pues es una tarea potencialmente larga que haría que la interface de usuario quedara bloqueada intermitentemente (además de hacer que nuestro proyecto dependiera de System.Windows.Forms).

La única solución factible, y la más elegante, es que haya otro hilo más, el hilo de Join, que permanezca a la espera hasta que exista información nueva de todas las cámaras.

Esta segunda barrera, a la que hemos llamado TimeManager, tiene un comportamiento distinto al gestionar este nuevo hilo y, además, tomar estadísticas del sistema de sincronización.

Es interesante mencionar el siguiente compromiso entre un diseño orientado a objetos puro y el rendimiento:

Puesto que es a partir de la reconstrucción 3D, inmediatamente después de la creación de rayos, cuando los problemas de sincronización se hacen visibles, parece que debiera ser RayosaPuntos3D quien se encargara del problema de sincronización atendiendo a las dependencias y a un diseño granular.

Sin embargo, como hemos visto, lo óptimo a nivel de rendimiento y calidad de resultados es minimizar el desfase entre la captura de *frames* de cada una de las cámaras, y esto se consigue anticipando la sincronización a la fase de Análisis 2D, en concreto a las responsabilidades de la clase BitmapToPuntos2D. Ésta segunda aproximación es la que hemos seguido.

3.3.4.2.3. Solución final

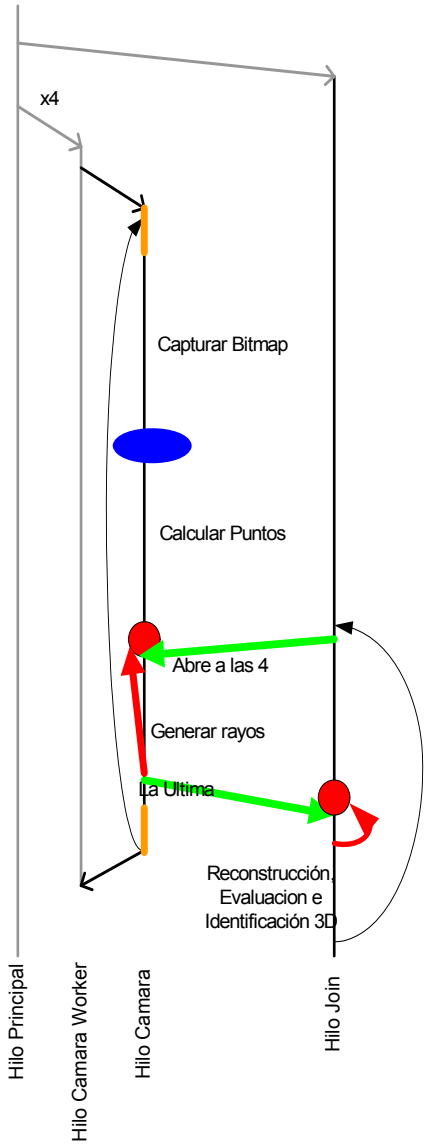
Para acabar de entender a nivel global el problema de la sincronización en la aplicación contabilizaremos el número de threads involucrados, y el papel que juega cada uno.

Como toda aplicación, el programa comienza con un **hilo principal**. Este hilo, puede ser una aplicación de windows, o el bucle principal de un juego a pantalla completa.

Cada cámara que utilicemos tiene, a su vez, dos hilos:

El **worker thread de la cámara** negocia la captura con DirectShow, y después queda bloqueado hasta que se pida interrumpir la captura. Este hilo es creado manualmente desde la clase CaptureDevice (dentro de .Net).

El **hilo de la cámara** es quien se encarga de generar los Bitmaps, y después lanza el evento que es aprovechado para el Análisis y Evaluación 2D, y la Construcción de Rayos. Opuestamente, este hilo es creado por DirectShow, que llama a la clase Grabber (anidada dentro de CaptureDevice) porque implementa la interface ISampleGrabberCB.



Finalmente hay un último hilo, el **hilo de join**, que es responsable de mezclar la información nueva de todas las cámaras y lanzar el proceso de reconstrucción, evaluación e identificación 3D.

En el siguiente diagrama se puede ver como el Worker Thread y el Hilo de las Cámaras se encuentra por cuadriplicado (suponiendo cuatro cámaras simultáneas).

La **linea amarilla** indica la región de código *unmanaged* que ejecuta DirectShow antes de llamar al Grabber.

La **elipse azul** muestra la posición de la barrera, inmediatamente después de la captura del Bitmap.

Los **círculos rojos** representan los EventWaitHandlers que tiene el TimeManager para sincronizar, siendo las **flechas verdes** las llamadas al Set(), y las **flechas rojas** la llamada a Reset().

3.3.4.3. Algoritmo pirámide

El algoritmo pirámide es el nombre que le hemos dado al algoritmo que realiza la reconstrucción de los puntos situados en el espacio (en 3 dimensiones) a partir de los rayos que tenemos de cada cámara.

La principal característica del algoritmo es que ha de ser tolerante a fallos, en el sentido de ser capaz de formar puntos sin que, necesariamente, haya un rayo de cada cámara que pase por él.

Así mismo, ha de ser simétrico: cualquier par de rayos de dos cámaras ha de ser capaz de formar un punto, sin dar una importancia especial a una cámara en particular.

Es un algoritmo que nos dará una mayor precisión de la posición de los puntos cuantas más cámaras tengamos, pero, por el contrario, su complejidad también será mayor a medida que aumentemos este número de cámaras.

Es un poco complicado, pues hace uso de varias estructuras de datos diseñadas *ad-hoc* para su funcionamiento. Para facilitar su entendimiento primero estudiaremos, a alto nivel, la idea del algoritmo, para definir con detalle más tarde las estructuras de datos y operaciones concretas que lo hacen posible.

3.3.4.3.1. Idea General

Como hemos dicho, el algoritmo trata de reconstruir Puntos3D a partir de RayosLuz de distintas cámaras.

En una primera aproximación, teniendo en cuenta sólo dos cámaras, el proceso es sencillo:

Se *cruzan* todos los rayos de una cámara (A) con los de la otra (B). Aquellos cruces de rallos que tengan un error pequeño (haciendo uso de la distancia entre rallos), son considerados PuntoLuz3D.

Al introducir una tercera cámara (C), la solución evidente es cruzar los PuntoLuz3D anteriores con los nuevos rayos. Ésta aproximación **no es viable**, pues es completamente asimétrica. De hacerlo así, sólo serían puntos los vistos por A y B, ayudando el resto de las cámaras pero teniendo un papel secundario.

En lugar de eso, se calculan el conjunto de puntos por triplicado: los de la cámara A con la cámara B, B con C y C con A.

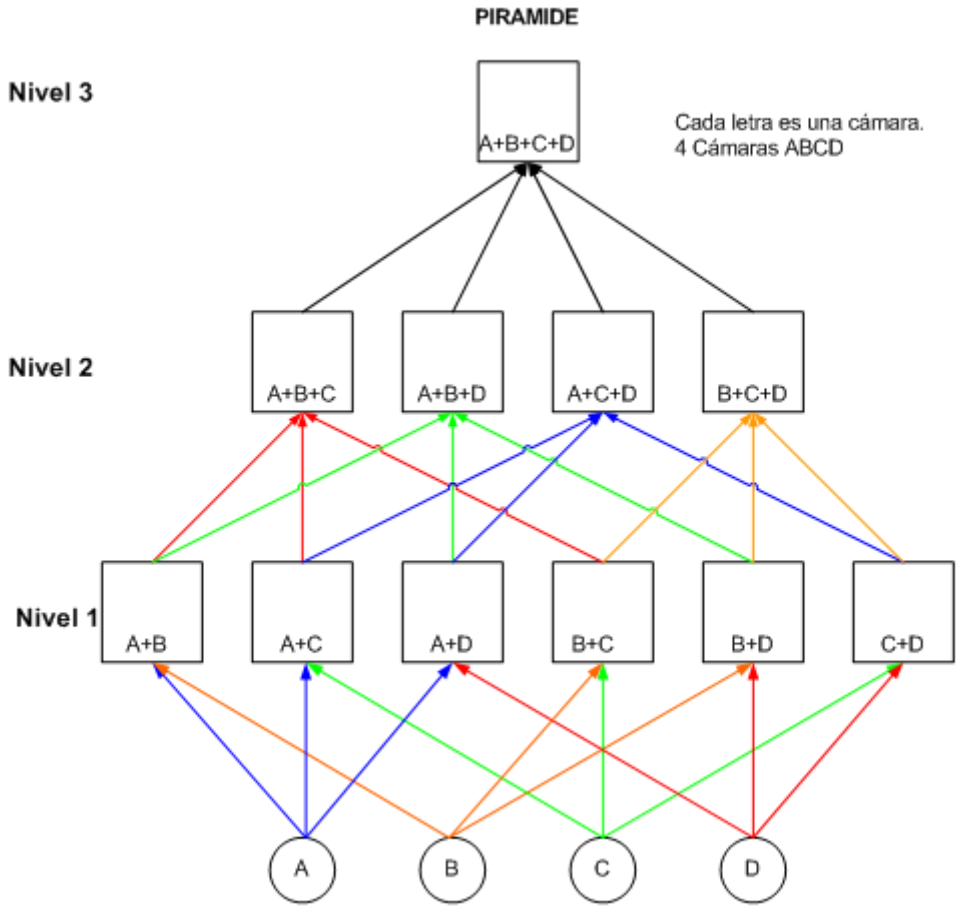
A cada uno de estos conjuntos de puntos intermedios le llamaremos tableros, y a los posibles puntos que contienen, casillas.

Finalmente, si existen casillas en los tableros A+B, B+C y A+C que están muy próximas entre sí, podrían ser unificadas a casillas de un tablero de orden superior, A+B+C, que contiene los puntos vistos simultáneamente por las tres cámaras.

Pero de haber algún problema con un punto, que no es visto por la cámara B, por ejemplo, seguiría existiendo su casilla en el tablero A+C, no perdiéndose sus datos.

Esta aproximación es mucho más simétrica y tolerante a errores.

Falta tan sólo generalizar la idea a **n** cámaras:



En el diagrama anterior se pueden ver las cámaras (círculos) y los tres niveles de tableros (cuadrados).

Los tableros del primer nivel son todos los cruces de cámaras posibles. Cada tablero recibe el nombre de las cámaras que lo componen. En el ejemplo hay 4 cámaras (ABCD) y 6 Tableros de Nivel 1, cada uno con dos cámaras, pues 6 es el número de combinaciones de 2 elementos en un conjunto de 4.

Los niveles siguientes, Nivel 2 y 3, tienen a su vez 4 y 1 elemento, pues:

Elementos Nivel 2 = $\binom{4}{3} = 4$

Elementos Nivel 2 = $\binom{4}{4} = 1$

De la misma manera, para saber qué tableros de orden inferior han de crear los tableros de orden superior, se utiliza de nuevo la combinatoria, por ejemplo:

Para conocer los tableros de Nivel1 (2 cámaras) que han de formar el tablero de Nivel2 ACD, se calculan las combinaciones de 2 elementos en ese conjunto, dando como resultado AC, AD y CD, que son las relaciones expresadas en el diagrama.

La idea es que para formar una casilla que relacione a 3 rayos, son necesarias las 3 casillas que relacionan a los 3 pares de rayos que se pueden hacer. Con 4 cámaras 4 tríos son necesarios… Y en general para hacer una casilla que relacione N rayos son necesarias las N casillas que relacionen a N-1 rayos cada una. Una idea similar se puede aplicar a tableros.

Ahora que ya tenemos una primera aproximación del funcionamiento del algoritmo, podemos explicar las particularidades de las estructuras de datos, y los detalles del algoritmo:

3.3.4.3.2. RayoLuz

Como sabemos, un RayoLuz es el resultado de proyectar un PuntoLuz2DEvaluado según la posición y orientación de una cámara. Esta proyección es realizada por la clase Camara3D.

Contiene un Ray (que define su origen y dirección), y la Camara3D y el PuntoLuz2DEvaluado que lo han creado.

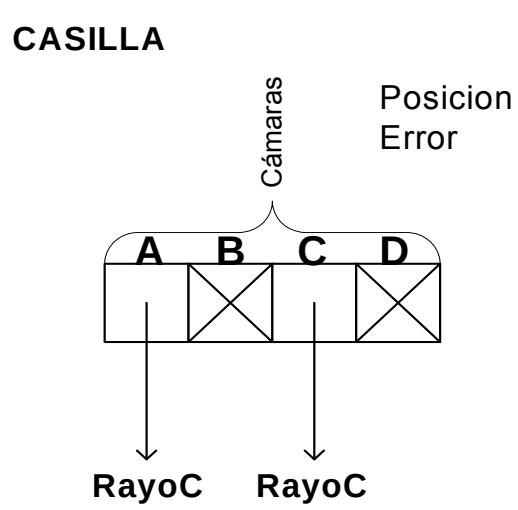
3.3.4.3.3. Casilla

Representa una posible unión de rayos, que puede dar lugar a un punto. Como veremos más adelante, una Casilla puede formarse a partir de dos RayoLuz de cámaras distintas, o mediante la unión de varias casillas.

Principalmente está definida por una posición en 3D y un error, pero tiene además otros campos para facilitar operaciones:

Tiene un array de RayosLuz, este array estará indexado por el id (un entero) de la cámara que ha capturado el rayo, y está dimensionado al número de cámaras existentes. De esta manera se simplifica la comprobación de si dos casillas son compatibles, un concepto que se verá más adelante.

Además, una casilla tiene una colección de casillas a partir de las cuales se ha obtenido (que será vacía si la casilla se ha obtenido por rayos y no por casillas), y un flag que nos indica si la casilla ya ha sido utilizada para construir otra casilla. De esta manera, a la hora de producir puntos desde casillas, cuando se consigue una casilla de nivel superior es fácil invalidar todas las casillas que la han dado lugar.



Una Casilla ha de tener al menos dos RayoLuz, es decir, el array ha de tener al menos dos posiciones ocupadas. Esto se debe a que una casilla representa si una serie de rayos se cruzan relativamente cerca (la cercanía la determinaremos en tiempo de ejecución, mediante el error), y para ello han de haber al menos dos rayos implicados.

Una Casilla se puede obtener de dos formas: A partir de dos RayoLuz, o mezclando dos o más casillas:
:

A partir de dos RayoLuz:

El primer método parte directamente de los RayoLuz de dos cámaras distintas, cuando aún no tenemos casillas creadas, y la casilla se inicializaría de la siguiente forma:

- Los dos RayoC se meterían en la posición que corresponda con su cámara en el array.
- Se calcula la intersección entre los dos rayos (visto en el apartado '3.2.4. Distancia entre rayos') lo que da como resultado dos puntosz (A y B) de intersección. El punto medio de estos dos puntos será la posición de la casilla, mientras que la mitad de su distancia (que coincide con la distancia del punto a cada rayo) al cuadrado (utilizamos distancias cuadradas) será su error.

A partir de dos o más casillas:

Para definir esta operación entre casillas es importante definir el concepto de **casillas compatibles.** Dos casillas son comparables si, al compartir la misma cámara, lo hacen por el mismo rayo.

Esta comprobación se puede hacer rápidamente gracias a que las casillas tienen, como hemos visto, un array indexado por id de cámara que nos indica el rayo.

Dadas N casillas de N-1 rayo cada una, todas compatibles con todas, se puede definir una casilla de N rayos de la siguiente manera:

- Los rayos que la componen son los rayos de cada cámara de todas las casillas que la forman, que no pueden ser contradictorios pues son todas casillas compatibles entre sí.
- La posición de la casilla es la media de las posiciones de las casillas que la forman, y su error el de recalcular la distancia cuadrada de la nueva posición a cada uno de los rayos que lo forman.

El objetivo final es poner todos los rayos que tenían las casillas anteriores en una única casilla pero con la condición de que si dos o más de estas casillas tienen un rayo en una posición correspondiente a una cámara, este a de ser común, o igual, en todas ellas.

Un detalle importante, no existe un método como el siguiente:s

```
public static Casilla FromCasillas(List<Casilla> casillas)
```

pues, desde el punto de vista de la clase tablero que sería su potencial usuario, le es dificil (e ineficiente) determinar que casillas incluir en la lista (que casillas son compatibles entre sí).

En lugar de eso la creación se divide en 3 métodos separados, que son llamados consecutivamente, y pueden dar finalmente el mismo resultado pero sin tener que determinar a priori las casillas compatibles:

```
public bool Mezclar(Casilla otra, out Casilla nueva)
```

Invocado sobre una casilla de N-1 rayos, y con otra casilla de N-1 rayos como parámetro, devuelve *'true'* si ambas con compatibles, y en ese caso devuelve por el parámetro de salida 'nueva' una nueva casilla de N rayos con los rayos de las dos anteriores.

```
public bool Compatible(Piramide piramide, Casilla otra)
```

Invocado sobe la casilla de N rayos recién creada (de la llamada anterior) y pasando una casilla de N-1 rayos por parámetros, devuelve *'true'* cuando ambas son compatibles.

```
public void NormalizarPosicionYCalcularError(int nivel)
```

Ajusta la posición y el error de una casilla completada para ajustarlos a su definición.

La motivación de esta estrategia, sin embargo, no acabará de entenderse hasta llegar a la clase Tablero.

3.3.4.3.4. Tablero

Para agrupar la colección de casillas vamos a utilizar una estructura de datos que hemos llamado Tablero. Esta estructura de datos tendrá como campos por un lado una lista con las cámaras de las que el tablero puede tener casillas y por el otro tendrá una lista de casillas.

Un tablero asociado a dos cámaras contiene, por tanto, todas las casillas que están asociadas a esas dos cámaras. Ésta organización hace mas sencillo encontrar casillas compatibles.

Al igual que las casillas, los tableros se pueden construir de dos formas: a partir de dos cámaras o a partir de una serie de tableros.

A partir de dos Cámaras:

Intenta construir casillas cruzando los rayos de las dos cámaras, y si se construye con éxito (el error es menor al error máximo establecido), la insertamos en las listas de casillas del tablero.

A partir de dos o más tableros:

Para formar un tablero a partir de otros, es necesario que estos cumplan ciertas restricciones, en concreto:

Para formar un tablero relacionado con N cámaras es necesario disponer de todos los tableros relacionados con las combinaciones N-1 cámaras de las N cámaras anteriores. Son exactamente N tableros.

Por ejemplo, para formar un tablero que relaciona a las cámaras ABC, son necesarios 3 tableros: AB, BC, y AC.

Salvadas estas restricciones, el problema consiste en construir casillas con la unión de las casillas de los otros tableros, e introducirlas en el nuevo tablero:

- En primer lugar cogemos todas las casillas del primero de los tableros y las intentamos **mezclar** con las del segundo tablero (explicado anteriormente en la sección Casilla).

- Seguidamente, comprobar cuales de estas casillas recién creadas son **compatibles** con las casillas del resto de los tableros.

- Cada casilla se normaliza: Se calcular la posición del punto que representa, y como error se pone la suma de las distancias cuadradas entre cada uno de los rayos que tiene la casilla y la posición del punto al que representa.

- Finalmente, se insertan en el tablero aquellas casillas que tengan un error por debajo del permitido, y se marcan como eclipsadas todas las casillas a partir de las que se ha obtenido la anterior casilla.

Debido al hecho de que un tablero es un objeto algo pesado y, sobretodo, que se conservan de una ejecución a otra (lo único que varían son las casillas de su interior), lo que haremos es *vaciar* cada tablero antes de cada ejecución pero conservando la estructura del tablero en sí.

Esta reutilización de los objetos tiene como repercusión el hecho de que, en lugar de los dos explicados anteriormente, tengamos que disponer de cuatro constructores distintos, pues pueden separarse entre **los que crean la estructura de datos**, y **los que la rellenan con casillas**.

3.3.4.3.5. Pirámide

La clase pirámide es la contenedora de los tableros, y también se preserva de una ejecución a otra. Es también la fachada del algoritmo de reconstrucción 3D.

Contiene un array con todas las cámaras que disponemos, y una '*tabla hash*' que asocia el **nombre normalizado** de un tablero con el tablero en sí.

Entendemos por nombre normalizado de un tablero, el string formado por las letras asociadas a las cámaras que lo componen, ordenadas de la manera alfabéticamente menor, por ejemplo: Para el tablero que asocia las cámaras C, D y A, su nombre normalizado es ACD.

Usando una '*tabla hash*' podemos obtener de forma inmediata los tableros a partir de las cámaras a las que está relacionada, lo que es muy útil para componer unos tableros a partir de otros.

Éstos son los métodos principales de la clase Pirámide:

Constructor:

Puesto que la estructura pirámide se preserva de una ejecución a otra, es necesario un constructor que genere el esqueleto de esta estructura de datos.

A este constructor se le pasan como parámetros las cámaras, asociándole a cada una letra identificativa (comenzando por la A).

Finalmente se construyen los tableros, inicialmente vacíos, por niveles: en el nivel 1 estarán los tableros que contienen dos cámaras, en el nivel 2 los que contienen 3 cámaras y así sucesivamente.

Finalmente se insertan estos tableros en la '*tabla hash*' con su nombre normalizado.

Calcular:

Se encarga de que todos los tableros generen sus casillas asociadas, llamando en el orden correcto (desde el nivel 1 al nivel superior) y con los parámetros correctos:

Para los **tableros de nivel 1**, los parámetros son las cámaras asociadas directamente.

Para el **resto de tableros**, los parámetros a reunir son los tableros de nivel inferior asociadas a las mismas cámaras excepto una, por ejemplo, para generar las casillas del tablero ABC son necesarios los tableros AB, BC y AC con las casillas ya generadas.

GetPuntos:

Genera una lista con los puntos en 3D que se puedan extraer de todas las casillas de esta pirámide.

Para ello recorre los tableros desde el nivel superior hasta el nivel 1 y, de cada uno, extrae las casillas que no hayan sido usadas para formar una casilla que esté en el nivel superior, para formar un PuntoLuz3D con ella.

Clear:

Vacía de casillas todos los tableros, para así poder reutilizarlos.

3.3.5. Evaluación de puntos 3D

Al igual que en los PuntoLuz2D, para los puntos 3D vamos a tener una nota que nos va a indicar lo *buenos* que son estos puntos.

Para considerar si un punto es bueno tendremos en cuenta tres criterios:

- **Distancia media de los rayos al punto:** Cuanto más pequeña sea esta distancia, de mayor calidad consideraremos el punto.
- **Distancia entre los colores de los rayos que lo forman:** Un PuntoLuz3D está compuesto por varios RayoLuz, que han sido deducidos a partir de PuntosLuz2D que tienen un color asociado. Tiene sentido, por tanto, exigir que estos rayos tengan un color similar para que el PuntoLuz3D sea válido. Mientras menor sea esta distancia mejor será el punto.
- **Número de cámaras del punto:** Mientras mas cámaras hayan visto a un punto (más rayos le atraviesen) mejor será el punto.

Utilizamos la siguiente ecuación para determinar si un punto 3D es suficientemente bueno:

$$\frac{\frac{\text{ErrorColor}}{\text{errorMaximoColor}}beta + \frac{\text{ErrorPosicion}}{\text{errorMaximoPosicion}}(1-beta)}{\text{NumCamaras}} < notaBuena3D$$

Están definidas por tanto 4 parámetros que han de ser configurados por el usuario:

errorMaximoColor: valor máximo que puede alcanzar el error color, normaliza el error color entre 0 y 1. Es un valor empírico.

errorMaximoPosicion: valor máximo que puede alcanzar el error de posición, normaliza el error de posición entre 0 y 1. Es un valor empírico.

beta: Gradúa la importancia relativa del color y la posición en la valoración final. Sirve como parámetro de interpolación.

notaBuena3D: Determina la cota, por encima de la cual, un PuntoLuz3D es filtrado.

Hay una pequeña diferencia con respecto a la nota de los puntos 2D, y es que la nota en los puntos 3D será mejor cuanto menor sea, mientras que en 2D es al revés. Esto se debe a que para calcular esta nota usamos los errores de posición y color, que son distancias y serán mejores cuanto menores sean.

3.3.6. Identificación de Puntos 3D

Identificar puntos significa darles identidad, y que esta se preserve de un frame a otro. Hasta ahora, aunque en todos los visualizadores nosotros mantenemos la identidad de los puntos 3D, realmente el proceso no conoce esta información. Para el algoritmo de reconstrucción no son más que coordenadas arbitrarias, tanto en 2D como en 3D, que recompone cada vez sin utilizar información del frame anterior. Si conoce, sin embargo, el color de los puntos 2D y 3D.

Para recuperar la identidad de los puntos nos hemos basado en el proceso que nosotros, los humanos, seguimos para preservarla de un *frame* a otro. Proceso que realizamos inconscientemente.

Hay sólo dos criterios que utilizamos para mantener la identidad de los puntos: Su color y su posición o trayectoria.

Su color se preserva para todo los frames. Un punto rojo pertenece a una luz roja y se puede confiar en que esta no cambiará a medio plazo. A corto plazo, sin embargo, si se producen destellos, reflejos u oclusiones que hacen que un punto sea percibido por alguna cámara como de un color objetivo incorrecto, lo que afecta al color final del Punto 3D. Finalmente, puede haber varios puntos de un mismo color, por lo que el color por si mismo no define la identidad.

Su posición, sin embargo, si varía continuamente. Es necesario considerar no sólo su posición, sino también su velocidad para predecir su posición en el siguiente frame. Adicionalmente, es muy frecuente que los puntos desaparezcan. **Estas desapariciones pueden ser cortas**, por una oclusión momentánea**, o tomar mucho tiempo**, por ejemplo cuando el usuario simplemente quita los puntos de la zona visible de las cámaras para utilizar el teclado o el ratón.

¿Que hace un humano en esta situación? Según nuestra aproximación lo que haría sería mantener la identidad basándose en la posición, excepto si el punto desaparece *demasiado tiempo*, en ese caso se basaría en los colores.

Con esta idea en mente construimos nuestro algoritmo:

- Inicialmente configuras el algoritmo para identificar un numero determinado de puntos, así como los colores que tendrán cada uno. Se pueden considerar estos puntos a identificar como *celda***.** Cada celda tiene un color asociado.

- A medida que nuevos puntos que han pasado la evaluación son aportados, se intentan colocar en las celdas de la manera más óptima posible. Éste número de puntos puede ser menor, igual o mayor que el número de celdas, por lo que es posible que sobren puntos o queden celdas vacías.

Esta definición del algoritmo es aún un poco vaga, queda por definir que se entiende por colocar puntos en las celdas y, sobretodo, que se entiende por una colocación óptima.

Dado que tenemos **c** celdas y **p** puntos, puede ocurrir:

- **c = p:** Tenemos el mismo número de celdas que de puntos, es el caso más simple y entendemos colocación por una permutación de c (=p) elementos.
- **c < p:** Si el número de puntos es mayor que el de celdas, es necesario generalizar a las **permutaciones de p en c.**
- **c > p:** Opuestamente, si es el número de celdas el que es mayor, valdría con generalizar, inversamente, a las **permutaciones de c en p.**

No es suficiente que con que contabilicemos el número de permutaciones que hay en cada caso, tenemos que ser capaces de generar dichas permutaciones para poder probarlas.

Definir que entendemos por colocación óptima es algo más complicado: El error global de una permutación es la suma del error de cada una de sus celdas al punto que le corresponde. Este error individual puede calcularse, a su vez, mediante una fórmula muy similar a la que utilizamos para evaluar los Puntos3D, con algunas salvedades:

$$\frac{\text{DistanciaColor}}{\text{distanciaMaximoColor}} alpha + \frac{\text{DistanciaPosicion}}{\text{distanciaMaximoPosicion}} (1 - alpha)$$

- La DistanciaColor es la distancia entre el color del Punto3D y el color definido para su casilla. No la distancia interna entre los colores de los rayos que forman un punto 3D.
- La DistanciaPosición es la distancia entre la posición del punto y la extrapolación respecto a las posiciones anteriores de los puntos que ocuparon esa casilla. No la distancia interna ente el Punto3D y los rayos que lo forman.
- Y sobretodo, alpha es una variable, mientras que beta es una constante.

Éste ultimo punto es el más importante. Alpha es un valor entre 0 y 1 que varía dinámicamente, y es el que permite balancear a que criterio hacer más caso: Al color o a la Posición.

Inicialmente su valor es 1, confiando solamente en el color pues no tiene información histórica de su posición.

De la misma manera, si perdemos de vista un punto, a la vez que intentamos extrapolar su posición vamos incrementamos exponencialmente el valor de alpha, acercándolo a 1, simulando así la pérdida de confianza en la DistanciaPosicion.

La parte más paradójica, sin embargo, ocurre cuando el punto permanece visible:

En ese caso el valor de alpha tiende a ajustarse al error del color. Es decir, si el error de color es bajo, entonces ¡dejamos de confiar en el color!, o al menos confiamos en la posición en mayor medida.

Esto permite identificar varios puntos del mismo color en el caso mejor, aunque evidentemente, si estos desaparecen por mucho tiempo será imposible reestablecer la identidad.

Además, si asumimos incorrectamente la identidad basándonos demasiado en la posición, y acabamos asignando un punto de un color a una casilla de un color distinto, la DistanciaColor será alta, y alpha irá incrementándose paulatinamente lo que en última instancia lleva a ignorar la información de la posición y basarse sólo en el color.

Este último efecto simula un hipotético '*enfado*' de una casilla, permitiendo que corrija una decisión inadecuada de manera tardía, cuando de otra forma sería difícil de corregir porque los dos puntos ya están muy separados.

En la explicación anterior hablamos de extrapolar la posición de las casillas dependiendo de su información histórica. Aunque en un principio utilizamos una extrapolación cuadrática para hacer esto, en la práctica no mejoraba el resultado. Finalmente utilizamos una interpolación lineal que tiene en cuenta los instantes en los que han sido capturados los puntos.

Igualmente, decimos que alpha tiende a 0, o tiende a la DistanciaColor. Para hacer esto utilizamos una recurrencia exponencial típica:

$$alpha_t = alpha_{t-1} * (1 - coefAprendizaje) + tendencia * coefAprendizaje$$

Siendo coefAprendizaje una constante determinada.

Finalmente, cada vez que el identificador recibe un nuevo conjunto de puntos y consigue identificarlos, lanza un evento que avisa a cualquier interesado que tiene la actualización de sus Puntos3Didentificados.

3.3.7. Loader

Para simplificar el uso de nuestro proyecto por aplicaciones de terceros hemos desarrollado la clase Loader. Ésta recibe un objeto Settings, serializable a XML, con toda la información necesaria para arrancar el servicio, además de un tipo enumerado que permite graduar hasta que punto arrancar el servicio: hasta Reconstrucción 3D, Evaluación o Identificación.

Así mismo dispone de dos eventos principales, PuntosRecienProcesados y PuntosRecienIdentificados.

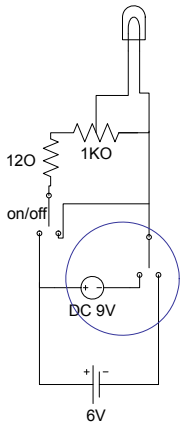
Finalmente, pero muy importante, loader es capaz de liberar todos los recursos, incluyendo los hilos, lo cual no es una tarea trivial.

3.4. Diseño Hardware

Como dijimos, nuestro proyecto está basado completamente en software para su funcionamiento. Es necesario, sin embargo, un pequeño diseño hardware para un guante con luces en los dedos, que será la principal interface con la que contará el usuario.

Los requisitos para el guante son los siguientes:

- Es posible alimentar las luces por 4 Pilas de 1,5 V o mediante un transformador de 9 V
- La intensidad de las luces ha de ser variable.
- El juego de luces ha de ser intercambiable, de manera que pueda usarse para un guante o para la plataforma de calibración.
- Debe tener un interruptor on/off



El diseño final es el sencillo circuito analógico explicado den la figura, y físicamente se encuentra alojado en un brazalete junto a las 4 Pilas de 1,5V.

La región rodeada por el círculo azul indica uno de los conectores del brazalete al transformador a 9V. El led, a su vez, es en realidad otro conector a un juego de luces intercambiables.

La resistencia de 12Ω es una resistencia de seguridad para no fundir los juegos de luces cuando el potenciómetro es llevado a su nivel más bajo

4. Aplicaciones

En esta sección hablaremos de las aplicaciones desarrolladas que utilizan nuestro HID Óptico 3D.

Al preferir acabar una aplicación de calidad con una base firme con la cual poder basarse en futuras ampliaciones, hemos centrado nuestras energías en la aplicación 'Configuración HID Óptico 3D'.

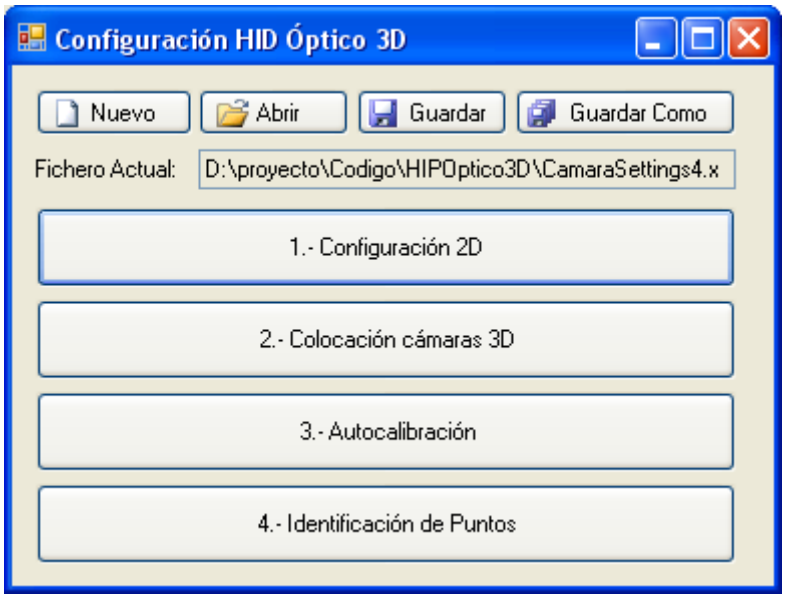
4.1. Configuración HID Óptico 3D

Esta aplicación sería la base para que terceros pudiesen utilizar la funcionalidad ofrecida por nuestro proyecto de una manera sencilla.

Se encarga de configurar todo el proceso de reconocimiento, reconstrucción e identificación de puntos, aislando en un archivo XML todos los parámetros de esta configuración. A la vez, expone de forma gráfica todo el funcionamiento interno del proyecto.

Es importante anotar que, a nivel de Interface de Usuario, la aplicación asume algunas restricciones con el ánimo de simplificar la configuración, como hacer que todas las cámaras utilicen la misma resolución, AspectRatio o FieldOfView. Estas restricciones no existen realmente en el Core.

La pantalla principal ofrece la funcionalidad común en una aplicación para trabajar con ficheros, en este caso, el fichero XML de configuración: Nuevo, Abrir, Guardar y Guardar Como.

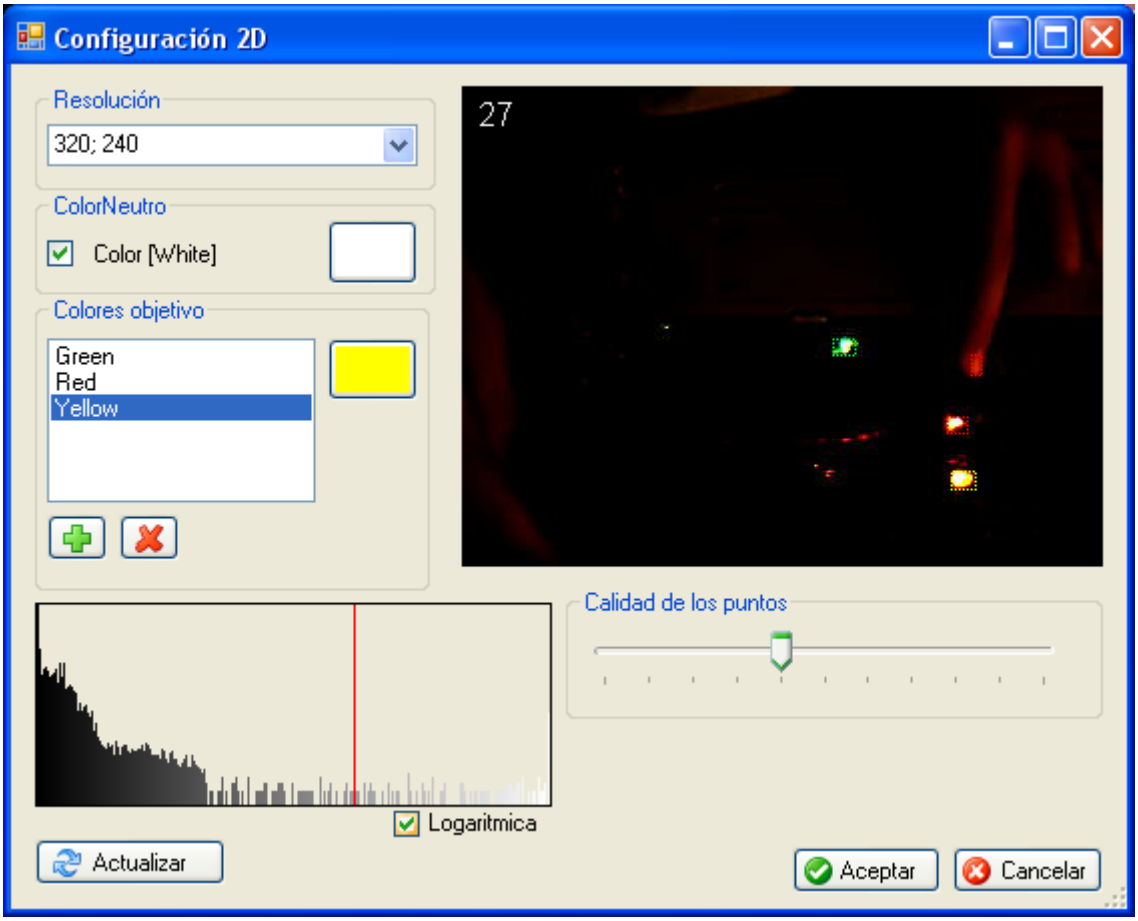


Expone, además, botones para navegar a las ventanas que configurará cada una de las etapas que componen el proyecto:

1- Configuración 2D
2- Colocación de Cámaras 3D
3- Autocalibración
4- Identificación de Puntos

Explicaremos cada una de estas ventanas a continuación:

4.1.1. Configuración 2D:



Desde esta ventana se pueden configurar los aspectos relacionados con las primeras etapas del proyecto: Captura de Imágenes y Análisis 2D.

La resolución es el único aspecto configurable de la etapa de **Captura de Imágenes**, además de las cámaras a utilizar que se configura en otro apartado. En este aspecto hay que mantener un compromiso entre calidad de la captura (resolución alta) y cantidad de ancho de banda USB disponible (resolución baja). En concreto trabajar a la mayor resolución (320 x 240) presenta problemas de ancho de banda al trabajar con más de tres cámaras.

En la etapa de **Análisis 2D** existen varios parámetros a configurar:

Para la Identificación de Puntos 3D es necesario determinar los colores de los puntos 3D a reconocer (colores objetivo), pero para que esto sea posible, ya en la etapa de análisis 2D es necesario tener el color en consideración. Desde ésta ventana se pueden determinar estos colores.

Como todos los puntos de luz comparten un color neutro en su centro (blanco), es conveniente activar el color neutro que es compartido por todos los puntos de luz.

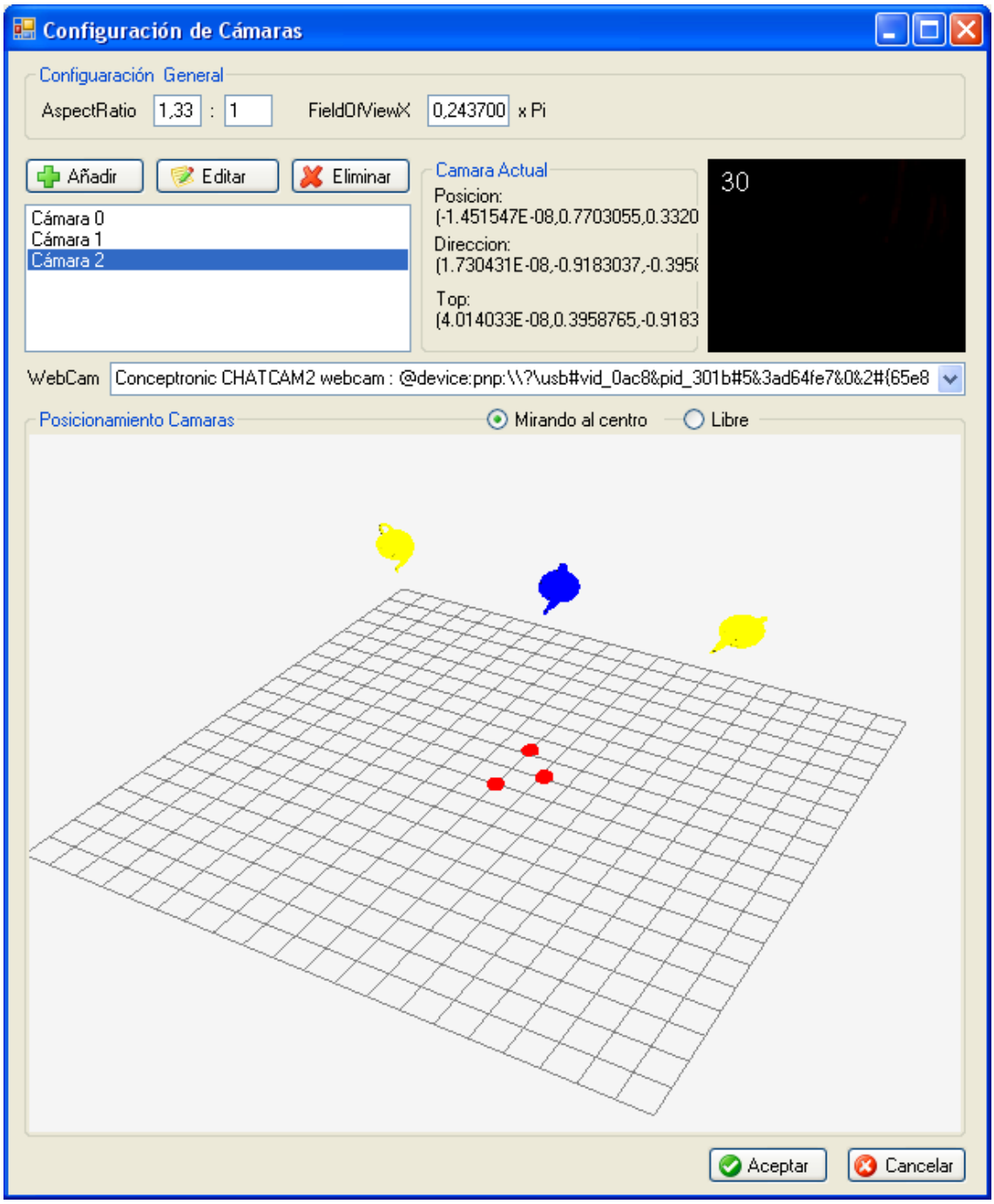
El Histograma nos permite ver de una manera global la distribución estadística de la luz en la imagen. Dependiendo de la situación de la raya vertical nos indicará a partir de qué valor consideramos un píxel suficientemente brillante siendo valorado como un posible punto.

Al realizar la gráfica de forma logarítmica se aprecia mejor la distribución de luminosidad en la imagen, pues de otra manera solo los colores negros puros son apreciables.

Finalmente, en la etapa de **Evaluación de Puntos 2D**, con el slider *'calidad de los puntos'* determinamos a partir de qué nota se considera si es un PuntoLuz3Devaluado es suficientemente bueno, siendo más restrictivos cuanto más a la derecha desplacemos el dispositivo deslizante.

En la esquina superior izquierda de la imagen capturada por la cámara aparece los frames/segundo que están siendo capturados.

4.1.2. Colocación cámaras 3D:



Desde esta ventana se pueden configurar los aspectos relacionados con las cámaras: número de cámara, posición y orientación de las mismas.

La cámara seleccionada se diferencia del resto de cámaras porque esta tiene color azul, frente al color amarillo de las restantes. Además debemos asociarle una WebCam física.

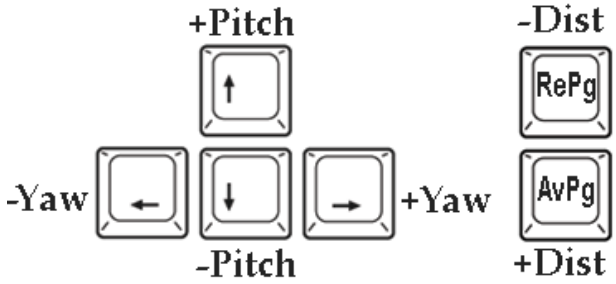
El posicionamiento de cada una de las cámaras se puede establecer de dos maneras: mirando hacia el centro, o mirando hacia cualquier dirección. Esta última opción les da mayor grado de libertad pero es más costoso configurarlas.

Dependiendo de la opción elegida, el teclado se comportará de distinta forma:

- Posicionamiento Cámaras: *Mirando al centro*
Todas las cámaras apuntan hacia el centro y giran en torno a dicho centro del eje de coordenadas.

Las distintas teclas posibles son las siguientes:

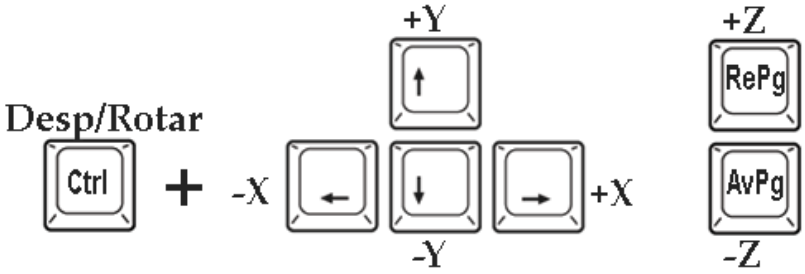
↑	Aumenta el pitch respecto al centro
↓	Disminuye el pitch respecto al centro
→	Aumenta el yaw respecto al centro
←	Disminuye el yaw respecto al centro
AvPg	Te alejas del centro
RePg	Te a cercas al centro



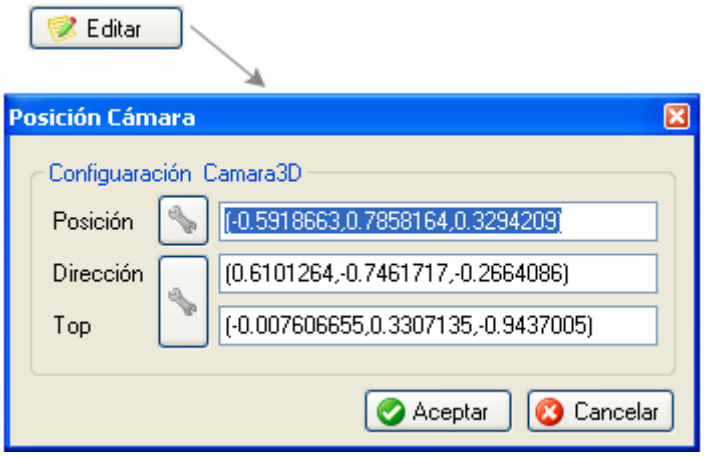
- Posicionamiento Cámaras: *Libre*
Libre posicionamiento de las cámaras.

Las distintas teclas posibles son las siguientes:

↑	Aumenta la coordenada y
↓	Disminuye la coordenada y
→	Aumenta la coordenada x
←	Disminuye la coordenada x
RePg	Aumenta la coordenada z
AvPg	Disminuye la coordenada z
Ctrl. + ↑	Aumenta la rotación sobre el eje y
Ctrl. + ↓	Disminuye la rotación sobre el eje y
Ctrl. + →	Aumenta la rotación sobre el eje x
Ctrl. + ←	Disminuye la rotación sobre el eje x
Ctrl. + RePg	Aumenta la rotación sobre el eje z
Ctrl. + AvPg	Disminuye la rotación sobre el eje z

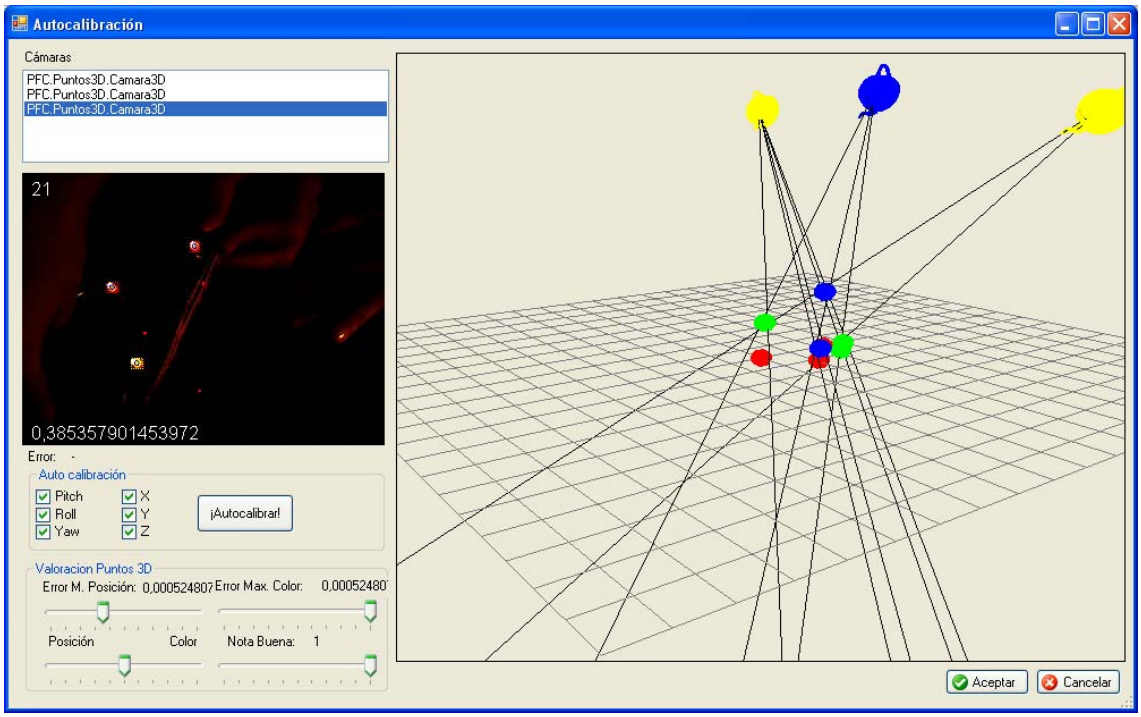


La visualización 3D posee tres puntos fijos (rojos) que nos sirven de marco de referencia para posicionar las cámaras, pero serán más usados en el punto '3.3.2.3. Autocalibración de cámaras'.



Aunque al añadir una cámara, esta ya posee su posición, dirección y top. Podemos editar su posición y cambiarla manualmente por valores que creamos que son más exactos.

4.1.3. Autocalibración de cámaras:



El formulario de Autocalibración de cámaras está orientado a que el usuario pueda corregir el error entre el posicionamiento virtual (establecido en el formulario anterior), y el real (dónde están las webcams físicas situadas realmente) de cada cámara.

Además, permite variar los parámetros para la evaluación de puntos 3D.

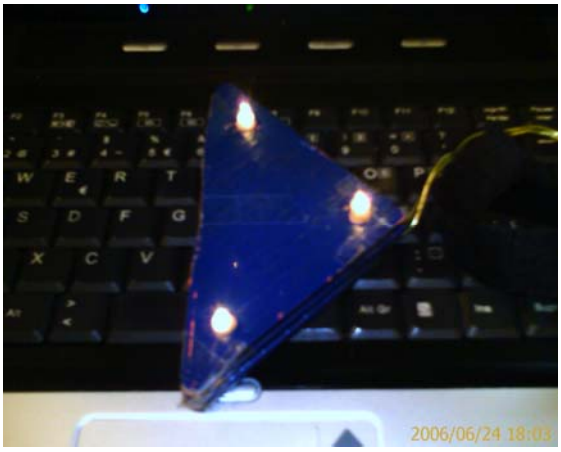
Al igual que en el punto *'4.1.2.Colocación cámaras 3D'*, la cámara seleccionada en *'Cámaras'* aparece de color azul en la visualización 3D. Sin embargo, en esta visualización se aprecian los Rayos de cada cámara y, en las intersecciones de estos, los PuntoLuz3D reconstuidos.

Además, al seleccionar una cámara se puede ver las imágenes que está capturando y, encima de ellas, unos pequeños puntos rojos. Estos puntos indican la proyección de los puntos rojos del mundo según la posición y orientación virtual de la cámara seleccionada.

Si, en el mundo real, colocamos un patrón similar de luces en el origen de coordenadas, debería detectarlos exactamente en la misma posición que los puntos rojos proyectados. Cualquiera error supone un desfase entre la colocación virtual y real de las cámaras. Para corregir este desfase tenemos dos posibilidades:

Corregir el posicionamiento real de la WebCam: Simplemente moviendo, con cuidado, la WebCam asociada a la cámara seleccionada hasta que los puntos detectados estén lo más cerca posible de los puntos proyectados (rojos).

Corregir el posicionamiento virtual de la cámara: Si se llega a un punto en el que mover la WebCam no es posible, o pensamos que el posicionamiento virtual no es correcto, podemos hacer que la cámara basándose en lo que ve, sea capaz de refinar los datos aportados por el usuario (esto es posible gracias al sistema de autocalibración explicado en el apartado '3.3.1.1.Algoritmos genéticos de auto configuración').



La manera de autocalibrar el posicionamiento de las cámaras es la siguiente: para realizar este paso, podemos prestarle mayor atención al pitch, roll, yaw, x, y o z dependiendo de lo que la vista nos de a entender que debemos mejorar. Es aconsejable autocalibrar pitch, roll y yaw y si con estos parámetros no mejoramos el resultado, utilizamos los parámetros x, y o z.

Dependiendo de cuántas cámaras vean cada uno de los puntos 3D, dichos puntos serán de diferentes colores:

- naranja: punto visto por cinco cámaras (intersección de cinco rayos)
- amarillo: punto visto por cuatro cámaras (intersección de cuatro rayos)
- verde : punto visto por tres cámaras (intersección de tres rayos)
- azul: punto visto por dos cámaras (intersección de dos cámaras)

Un punto es la intersección de dos rayos, por lo que es necesario como mínimo que dos cámaras vean al mismo punto.

Cuando los puntos rojos y los puntos 2D que ve la cámara coincidan para todas las cámaras, deberemos ver en la visualización 3D que los puntos que aparecen en la intersección de los rayos coinciden con los puntos fijos rojos.

En la imagen capturada por la WebCam podemos observar cómo, además de mostrar los frames/segundo que son capturados, nos muestra el error del algoritmo evolutivo:

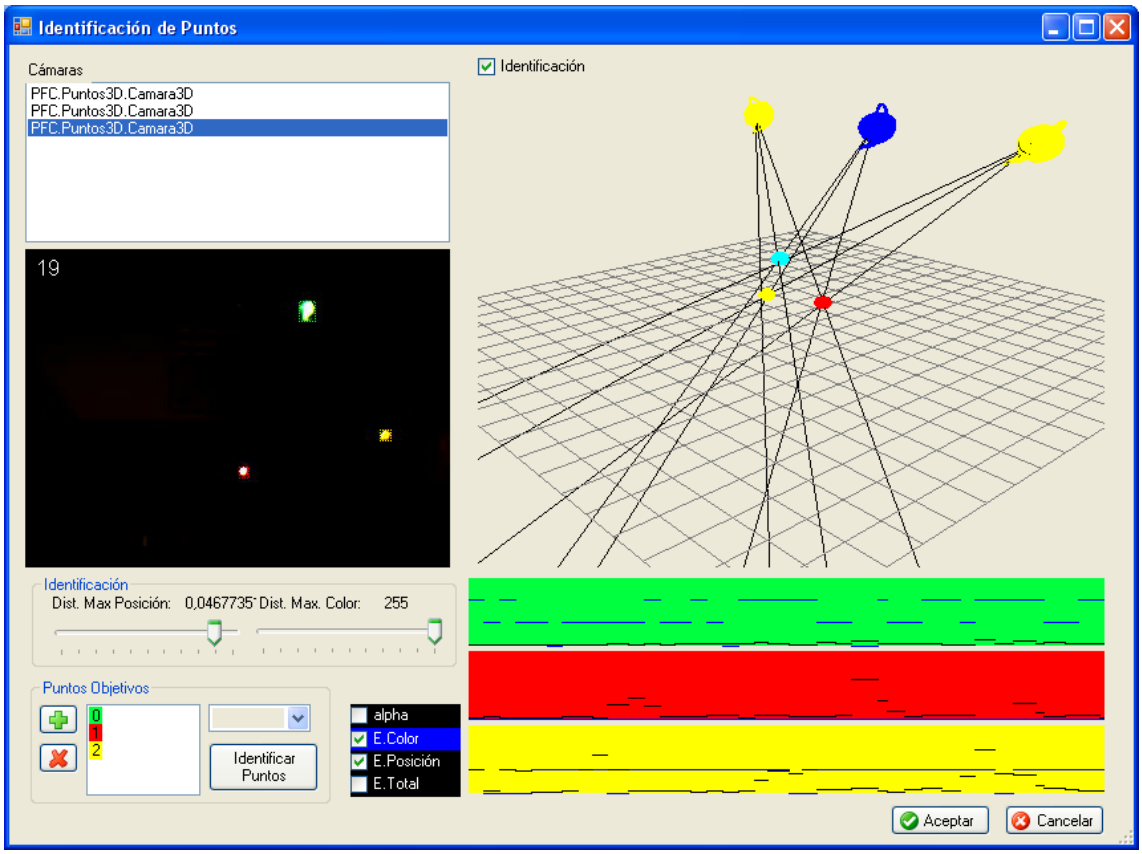
la suma de distancias cuadradas entre los puntos proyectados y los puntos vistos por las WebCams.

Finalmente, este formulario nos permite también ajustar los parámetros para la evaluación de puntos 3D. (Explicada en el apartado 3.3.2.3. Evaluación de puntos 3D) mediante sliders:

- ErrorMaximoColor
- ErrorMaximoPosicion
- Beta (Posición – Color)
- Buena Nota

Dependiendo de los cambios que hagamos en estos parámetros, variaremos los criterios de evaluación, y la nota de corte, reflejándose en la visualización 3D al aparecer más o menos puntos en la intersección de los rayos.

4.1.4. Identificación de puntos:



Desde esta interface podemos configurar la última etapa del proceso: la identificación de puntos. Debemos indicar cuántos puntos queremos identificar, y de qué color serán dichos puntos. Los colores objetivo disponibles serán los mismos que los que configuramos anteriormente en el punto 4.1.5. Configuración 2D. Pero en este caso el orden en el que pongamos estos colores determinará el índice de cada punto, y los colores pueden ser repetidos, representando dos puntos del mismo color.

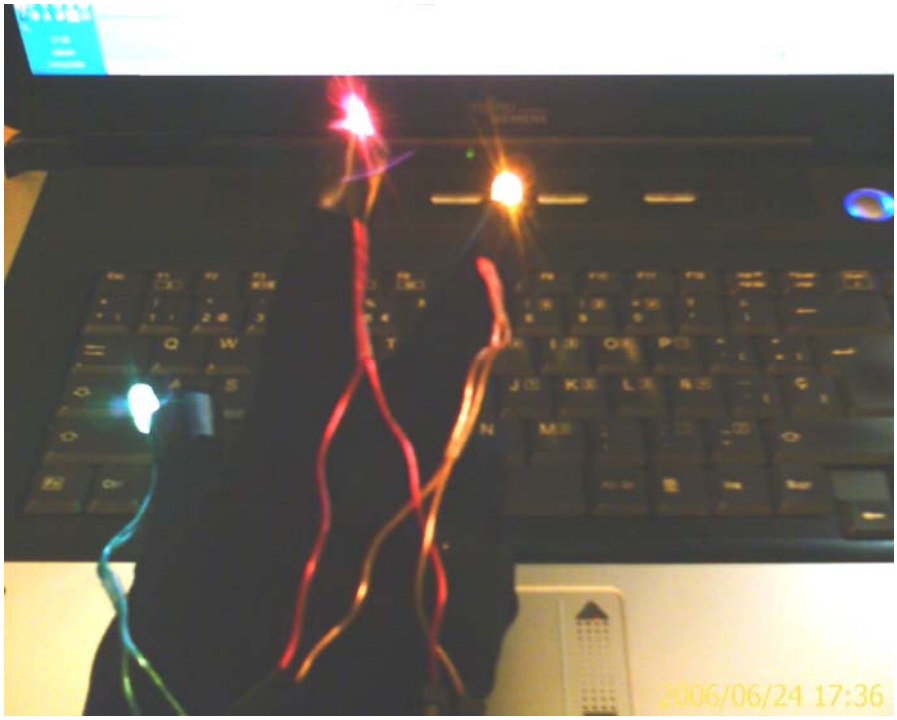
Como ya habíamos explicado en el punto 3.3.6. Identificación de puntos 3D, para que la identidad se preserve de un frame a otro usamos dos criterios: la DistanciaColor y la DistanciaPosición. Dichos dos criterios van de la mano con la variable alpha (de 0 a 1) que nos indica si debemos 'hacer más caso' a la DistanciaColor o por el contrario debemos tender a la DistanciaPosición. Además de estos tres parámetros que se pueden observar en la tabla existe la media entre la DistanciaColor y DistanciaPosición denominándola DistanciaTotal.

Cómo pueden observarse, en tiempo real se genera una gráfica que indica el valor de estas cuatro variables para cada uno de los puntos identificados con el color objetivo correspondiente.

Como ya ocurría en el punto *'4.1.3.Autocalibración de cámaras'*, la cámara seleccionada en *'Cámaras'* aparece de color azul en la visualización 3D. Además al seleccionar una cámara se puede ver las imágenes que está capturando. En el visualizador 3D, sin embargo, existe una casilla 'Identificación' que permite cambiar el modo de funcionamiento:

- Desactivada: Se muestran todos los puntos, su color depende del número de rayos que lo atraviesan, igual que en el punto anterior.
- Activada: Sólo se muestran los puntos identificados, y de no encontrarse, la extrapolación de éstos.

Además, existen dos sliders para la Distancia Máxima Color y la Distancia Máxima Posición, que sirve de cota para la variación de la posición y el color de un punto 3D de un frame a otro. Si dichos sliders toman un valor alto la nota de dichos puntos 3D será más pequeña, mientras que si el valor de los sliders es pequeño dichos puntos 3D tomarán valores más altos. Es importante tener en cuenta que cuánto más pequeña sea la nota de los puntos en 3D mejor valorado estará, lo que influye en la permutación escogida para identificar los puntos. El comportamiento está mejor explicado en explicado *'4.1.5. Identificación de Puntos 3D'*.



En este dibujo se puede observar como tenemos un guante con tres puntos objetivo, cada uno de un color diferente. En el punto *'3.4 Diseño Hardware'* mostramos el circuito del guante.

4.2. Rocket Commander

Además del programa de configuración del HID, que desarrollamos además con ánimo de explicar el funcionamiento del proyecto, integramos en el juego Rocket Commander la capacidad de ser manejado mediante puntos de luz.

Rocket Commander es un videojuego escrito por Benjamin Nitschke en C#. Pertenece a la iniciativa Code4Fun de Microsoft por lo que su código fuente está abierto y existen multitud de tutoriales y videos que explican su funcionamiento. Además hace un gran uso de las últimas técnicas de *shaders* 3D con HLSL lo que le da un aspecto muy atractivo.
(http://msdn.microsoft.com/coding4fun/gamedevelopment/rocketcmd/default.aspx)
(http://www.rocketcommander.com)

Todo esto hace de Rocket Commander un juego perfecto para integrar nuestro proyecto en él.

La integración es muy sencilla, primeramente se configura desde la aplicació anterior todos los parámetros de funcionamiento, incluidos los puntos a reconocer.

En nuestro caso queríamos que el cohete se manejara moviendo el dedo índice, con dos luces, la roja delante y la verde atrás. Por tanto hay que determinar estos dos puntos como puntos objetivos.

Ya en el código, lo primero es cargar el Loador con el archivo e configuración XML, y arrancar el sistema en su totalidad (hasta Identificación3D).

Para traducir la información del ángulo del dedo a coordenadas de la pantalla, que es lo que Rocket Commander se espera como interface de ratón, restamos las coordenadas de los dos puntos localizados (PuntoRojo – PuntoVerde), y de éste vector extraemos Pitch y Yaw.

Utilizamos estos dos valores como el incremento de la Y y la X respectivamente, por tanto si el dedo índice está perpendicular a la pantalla (Pitch = 0 y Yaw = 0) se obtiene el mismo efecto que si no moviéramos el ratón. Éste comportamiento también se da cuando ningún o sólo un punto son visibles.

Finalmente para mejorar el rendimiento y hacer una *sincronización debil* entre el juego y el sistema de captura, forzmos a hacer un Thread.Sleep(0):

- Antes de pedir las coordenadas en el Thread del juego.
- Después de generar las coordenadas en el Thread de captura de puntos.

Esto tiene el efecto de mejorar la sincronización entre los dos hilos, siendo la mayoría de las veces 1 captura de puntos = 1 ciclo del juego.

5. Bibliografía

http://www.codechamber.com/sharpgl/home.php
http://csgl.sourceforge.net/
http://www.ekampf.com/Sharp3D.Math/
http://www.codeproject.com/cs/media/Motion_Detection.asp
http://www.haddd.com
http://www.codeproject.com/cs/media/floodfillincsharp.asp
http://www.codeproject.com/gdi/QuickFill.asp
http://es.wikipedia.org/wiki/Arquitectura_de_software
http://www.artima.com/intv/simplexity.html
http://en.wikipedia.org/wiki/Coupling_%28computer_science%29
http://en.wikipedia.org/wiki/Inversion_of_Control
http://en.wikipedia.org/wiki/Graphics_pipeline
http://www.microsoft.com/spanish/msdn/estudiantes/eventos/imaginecup2006.asp
http://www.codeproject.com/gdi/QuickFill.asp
http://www.albahari.com/threading/
http://www.joelonsoftware.com/articles/LeakyAbstractions.html
http://www.microsoft.com/spanish/msdn/estudiantes/eventos/imaginecup2006.asp

6. Lista de palabras para búsqueda bibliográfica

1-Optical Motion Capture (MOCAP)
2-WebCam
3-HID
4-Quick Fill y Food Fill
5-Spherical Linear Interpolation (Slerp)
6-Quaternion
7-Algorítmo genético
8-C#
9-Geometría
10-Frustum