_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE UPV INGENIEROS DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Control de cámara de vídeo 3D.

Aplicación para selección de parámetros

Antonio de Román Martínez

Tutor: José Manuel Mossi García

Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Graduado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Curso 2015-16

Valencia, 1 de julio de 2016

Resumen

La presente memoria, recoge mi propuesta de trabajo final de grado. stereoTuner es una aplicación de calibrado y procesado en 3-D con una interfaz gráfica basada en la librería GTK, cuyo objetivo es elaborar un mapa de disparidad lo más adecuado posible, partiendo de un par de imágenes tomadas por una cámara estéreo. En primer lugar, después de una breve introducción al trabajo y la enumeración de los objetivos del mismo, introduciré los conceptos básicos de la visión binocular desde un contexto puramente fisiológico. En segundo lugar, introduciré y detallaré los conceptos matemáticos (desde el punto de vista de un sistema óptico) de la visión estéreo, así como los algoritmos seleccionados para la búsqueda de la correspondencia en imágenes estéreo. Una vez introducidos los fundamentos técnicos esenciales, introduciré y desarrollaré por secciones, el desarrollo del trabajo. Después de una breve introducción, describiré la estructura de la interfaz. A continuación, detallaré las tecnologías y librerías seleccionadas para su desarrollo, así como algunos conceptos importantes de la configuración de la aplicación. En tercer y último lugar, detallaré todas las funcionalidades implementadas. Por último, detallaré mis conclusiones y algunas líneas de trabajo futuro.

Resum

La present memòria, arreplega la meua proposta de treball final de grau. stereoTuner és una aplicació de calibrat i processat en 3-D amb una interfície gràfica basada en la llibreria GTK, l'objectiu de la qual es elaborar un mapa de disparitat el mes adequat possible, partint d'un parell d'imatges preses per una càmera estèreo. En primer lloc, després d'una breu introducció al treball i l'enumeració d'objectius del mateix, introduiré els conceptes bàsics de la visió binocular des d'un context purament fisiològic. En segon lloc, introduiré i detallaré els conceptes matemàtics (des del punt de vista d'un sistema òptic) de la visió estèreo, així com els algoritmes seleccionats per a la busca de la correspondència en imatges estèreo. Una vegada introduïts els fonaments tècnics essencials, introduiré i descriuré, el desenvolupament del treball. Després d'una breu introducció, descriuré l'estructura de la interfície. Després, detallaré les tecnologies i llibreries seleccionades per al seu desenvolupament, així com alguns conceptes importants de la configuració de l'aplicació. En tercer i últim lloc, detallaré totes les funcions implementades. Finalment detallaré les meues conclusions i algunes esbosses de treball futures.

Abstract

The following report is a summary of my undergraduate thesis proposal. stereoTuner is a 3D calibration and processing application, with a graphic interface based on the GTK library, the objective of which is to generate a disparity map which is as accurate as possible, given two images created by a stereo camera. Following a brief introduction to the thesis and list of the objectives of the same, the basic concepts of binocular vision, from purely physiological basis, are explained. Then, mathematical concepts of stereo vision (from the point of view of an optical system) are detailed, as well as algorithms used in the stereo imaging matching process. Once the technical basis are explained, the development of the project is set out in various sections. First of all, the structure of the interface is described. Secondly, the technologies and libraries selected for the project are detailed, in addition to selected concepts relevant to the configuration of the application. Thirdly, all of the implemented features are described. Finally, my conclusion are out forward, as well as some proposal for future work.

Agradecimientos

Antes de comenzar, me gustaría agradecer a todas aquellas personas que me han acompañado y ayudado durante la carrera:

A mis padres y hermano, por el esfuerzo, dedicación y tiempo que han depositado en mí. A mi pareja, que con su apoyo, ayuda y comprensión, me ha mantenido fuerte y con confianza a lo largo de esta etapa. A mis compañeros, que me han acompañado y ayudado durante estos años. A mis amigos, que siempre que los he necesitado, han estado ahí para apoyarme y ayudarme en todo lo posible. Y por último, a mis profesores, en particular a mi tutor, que sin su ayuda, constancia y paciencia, esto no habría sido posible.

Índice

Capítulo 1. Introducción del TFG

stereoTuner es una aplicación de calibrado y procesado 3-D, cuya idea original fue propuesta por Martín Peris, aunque fue de la adaptación de Marcos Martínez O'Kelly desde la que partí para desarrollar la aplicación actual. El esqueleto del cual partí (figura 8), constaba de una única pantalla y se limitaba al cómputo del mapa de disparidad a partir de dos imágenes estéreo. La inclusión de ficheros de configuración y calibración, nuevas pantallas, aplicación de estilos gráficos, barra de herramientas, una larga lista funcionalidades, optimización del código, inclusión de patrones de diseño… En definitiva, todo lo que voy a describir a continuación, es un desarrollo propio y es mi propuesta de trabajo final de grado.

Aunque en el siguiente capítulo, he enumerado los objetivos principales de partida para la realización del trabajo, es correcto indicar, que el objetivo primordial de la aplicación es que el usuario obtenga un mapa de disparidad lo más adecuado posible y para ello, se necesitan herramientas. Si nos limitamos al uso de las funciones de **OpenCV** que implementan los algoritmos de correspondencia, no estamos optimizando la obtención del mapa, es más, ni siquiera podríamos saber si los datos obtenidos son correctos. De ahí que se hayan desarrollado herramientas como las barras de rectificación, un área de información para ver las coordenadas exactas y la disparidad o profundidad calculada en ese punto, la sobreimpresión de mapas de color en función de la disparidad, la herramienta Capture, la cual nos permite verificar mediante una interfaz amigable con supervisión humana la disparidad en un punto exacto de la imagen, la posibilidad de guardar los ajustes predefinidos (tanto de los datos algorítmicos como de otras muchas propiedades) en ficheros de configuración y muchas otras funcionalidades que detallaré a partir del punto 3.3.

Vivimos en un mundo en el que la tecnología forma parte de nuestro día a día y eso hace que la competencia en el mundo profesional sea cada vez más alta. No sólo es importante hacer buenas aplicaciones (funcionalmente hablando), sino también hacerlas atractivas para los usuarios, por lo que es muy importante hacer especial hincapié en la estética y facilidad de uso de las aplicaciones que se vayan a desarrollar. Yo me considero una persona perfeccionista y sabedora de que entrar por los ojos de los consumidores, es allanar mucho el terreno hacia su fidelidad comercial. Por ello, he dedicado bastante tiempo a mejorar estéticamente la aplicación.

Por último, es importante que antes de abordar la estructura, tecnologías, configuración o las funcionalidades de la aplicación, repasemos los fundamentos fisiológicos y matemáticos de la visión binocular, a fin de comprender mejor la motivación del trabajo (puntos 3.1 y 3.2).

Capítulo 2. Objetivos del TFG

2.1 Objetivo principal

El objetivo principal de este proyecto y por ende, de la aplicación, es el de elaborar un mapa de disparidad lo más adecuado posible partiendo de un par de imágenes estéreo. Si bien este sería el objetivo principal del programa, hay muchas otras propuestas u objetivos, que se propusieron y que ayudan a conseguir y a mejorar este objetivo principal. En este capítulo, nombraré los objetivos iniciales que se propusieron antes de comenzar la fase de desarrollo, los objetivos secundarios que propuse durante dicha fase y los objetivos finales que propuse al terminar el desarrollo, lo que correspondería a las fases de test, producción y documentación.

2.2 Objetivos iniciales

Antes de comenzar a desarrollar la aplicación, se propusieron unos objetivos iniciales, que a medida que se iban superando, se iban ampliando. Inicialmente se propusieron los siguientes:

- Un área de botones e información para:
    - Activar o desactivar las rayas paralelas en las imágenes (líneas de rectificación).
    - Activar o desactivar que la imagen de profundidad sea una capa de color sobre la imagen de grises izquierda.
    - Que al mover el ratón sobre la imagen de profundidad aparezca la medida en números en la zona de botones.
    - Que los botones sean pequeñas imágenes nemónicas de la función que realizan.
    - Elegir que las imágenes estén ya rectificadas o que sean sin rectificar y el programa las rectifique con un fichero de rectificación.
    - Un botón para mostrar ayuda y que aparezca un texto donde se expliquen los parámetros, rango de valores y efecto de aumentar o disminuir.
- Que la imagen de disparidad aparezca a la izquierda en vez de a la derecha, debido a que la sobreimpresión de color se realiza sobre la imagen izquierda y de esta manera, es más sencillo ver la relación.
- Que la imagen izquierda y derecha aparezcan con muy poca separación entre ellas.
- Que la imagen de disparidad sea coloreada en función de la disparidad o de la profundidad.
- Para determinar la disparidad mínima y el número de disparidades, ver manualmente sobre el par de imágenes cual es la disparidad real que hay en el punto de la escena más alejado y el más próximo.
- Revisar el *display* de la imagen de disparidad, debido a que se observan cambios de brillo importantes, podría ser debido a la normalización de CImgDisplay.
- Revisar cuando la ventana principal se reduce en tamaño a petición del usuario, llega un momento que parte de las imágenes y objeto se ocultan. Mirar si se puede poner límite inferior para cuando se arranca por defecto y que el tamaño mínimo garantice que las

imágenes se muestren por completo y luego no se pueda hacer más pequeña, o bien que se re-escalen, pero que quede todo mostrado.

2.3 Objetivos secundarios

Como he mencionado en el punto anterior, inicialmente se propusieron unos objetivos y a medida que fui desarrollando, fui incorporando algunos más. Secundariamente, propuse los siguientes:

- Carga de valores predeterminados al iniciar la aplicación, de modo que cuando el usuario salga de ella y vuelva a acceder, algunas de las propiedades continúen exactamente igual.
- Habilitar el área de coordenadas y disparidad para todas las imágenes.
- Un botón que permita ver los valores en función de la disparidad o de la profundidad.
- Inclusión de un panel a modo de archivador por pestañas, en la parte inferior derecha de la ventana principal, para seleccionar entre varios menús o funciones.
- Inclusión de una pestaña con una lista que informe cronológicamente de las acciones llevadas a cabo por la aplicación, así como de posibles errores que se puedan producir (*logger*).
- Estilos gráficos propios de la aplicación, distribuidos uniformemente por todas las pantallas.
- Inclusión de nuevas ventanas, para no cargar la ventana principal:
    - Una ventana de preferencias, dónde se puedan controlar los parámetros que se informan en el fichero de configuración y que se pueden modificar y grabar desde este punto.
    - Una ventana que permita seleccionar varios tipos de archivos del sistema.
    - Una ventana que permita seleccionar de entre todos los mapas de color disponibles.
    - Una ventana que permita crear o seleccionar los ficheros de configuración (perfiles de configuración).
    - Una ventana que muestre, algunas pistas acerca de la aplicación (ventana de ayuda rápida).
    - Una ventana dónde se visualice uno o varios vídeos de ayuda, para las funciones más complejas de la aplicación.
    - Una ventana desde la que se pueda exportar la información del *logger*.
    - Una ventana para poder crear ficheros de calibración.
- Inclusión de ficheros de calibración, propios para cada algoritmo que se implemente y que guarden los datos de los ajustes del panel de calibración.
- Posibilidad de crear tantos ficheros de calibración como se desee, a la par de poder cargarlos durante la ejecución del programa, sin necesidad de tener que salir o realizar ninguna recarga estática de contenido (realizar una recarga dinámica).
- Disponer de validaciones genéricas para todas las entradas de texto y selectores de archivo que verifiquen:
    - Que la extensión del archivo seleccionado es la adecuada según el sitio desde el cual se abra.
    - Que la cadena introducida no contenga caracteres no imprimibles por los principales SO.
    - Que en caso de fallo se muestren mensajes de error con la validación, independientes para cada caso.
- Elaboración de una librería de etiquetas **xml** propias para la aplicación, así como el desarrollo de un *parser* para extraer su contenido. Distribuyendo estas etiquetas entre los ficheros de configuración y calibración generados para la aplicación. Permitiendo no sólo su lectura, sino también su escritura.

2.4 Objetivos finales

Una vez terminado el desarrollo de la aplicación, propuse algunos objetivos finales respectivos al test y documentación de la aplicación. Finalmente se propuso lo siguiente:

- Estructuración y optimización del código según funcionalidades. Una estructura de directorios coherente y que haga referencia a su contenido.
- Eliminación de código redundante o repetido, así como el añadido de bloques de comentario para relatar las partes de código más complejas.
- Añadido de *logs* en todas las funciones y acciones importantes en el código, así como en las validaciones (sobre todo en las no satisfactorias), para llevar un control exhaustivo de todas las actuaciones del usuario, así como para informar de los posibles errores o contratiempos durante la ejecución de la aplicación.
- Mejora estética de algunos aspectos secundarios de la aplicación: igualdad de márgenes, igualdad de tamaños en ciertas ventanas, mismos colores para los elementos que sean idénticos, etc.
- Grabación de varios vídeos de ayuda para los futuros desarrolladores (proyectistas) de la aplicación, para facilitar la comprensión del código desarrollado y que puedan comenzar a trabajar en los nuevos desarrollos con celeridad.
- Búsqueda de futuros desarrollos y líneas de actuación futuras según mi conocimiento de la aplicación. Qué cosas creo que se podrían mejorar, qué cosas se podrían ampliar, qué estaría bien incorporar o qué metodologías, tecnologías u otros aspectos incrementarían el valor de la aplicación.

Como objetivo final, me propuse la redacción de una documentación técnica (anexada al documento) para enumerar los métodos de cada fichero del código, así como estructuras, variables, etiquetas, patrones, etc. En el *stereoTuner Reference Manual* (7.3 [5]), existe una enumeración de todos los ficheros del código, una breve explicación de qué hace cada método, los parámetros de entrada y si procede, la salida del método, así como todos lo estrictamente necesario para conocer técnicamente el código de la aplicación. Este documento, se realiza con el objetivo principal de dotar de conocimientos al futuro desarrollador y se le insta a actualizarlo a la par que desarrolle en la aplicación, para que de este modo, cualquier persona que inspeccione o trabaje en el código, pueda tener un conocimiento previo de sus características. Aunque si bien es cierto que este manual no aparece en este documento, su realización también forma parte del proyecto.

Capítulo 3. Desarrollo del trabajo

3.1 Fundamentos fisiológicos de la estereopsis

3.1.1 Introducción

Se define la visión estereoscópica o estereopsis, como el fenómeno por el cual, nuestro cerebro es capaz de procesar un par de imágenes proyectadas en la retina de cada ojo y transformarlas en una sola imagen tridimensional. La mejor manera de entender el funcionamiento del ojo, es entendiéndolo como un sistema óptico convergente cuyo objetivo es formar una imagen del exterior invertida, sobre la capa sensible de la retina. Este proceso se conoce como proceso de formación de la imagen y es la primera fase de la visión, culminando con la imagen final con la que somos capaces de percibir la profundidad de cada punto situado dentro de nuestro campo visual (3.1.10).

3.1.2 Estructura anatómica

El ojo sigue una estructura basada en capas, dentro de cada cual encontramos distintos elementos y fluidos que realizan funciones concretas. Las capas que componen el ojo son las siguientes:

- Capa externa: En la capa anterior se encuentra la esclera, este tejido fibroso funciona a modo de escudo para proteger la córnea (que también se encuentra en la capa externa) a través de la cual pasa la luz.
- Capa media: Esta capa está compuesta por la úvea, formada por el iris, la coroides y el cuerpo ciliar. El iris es uno de los elementos más importantes del ojo, siendo el encargado de regular la apertura (regulando la entrada de luz al sistema), así como el cuerpo ciliar, que lo es para el proceso de acomodación.
- Capa interna: Formada por la retina, ésta está directamente conectada al sistema nervioso central (por ende al cerebro) mediante el nervio óptico.

El interior del ojo se divide en tres compartimentos:

- Cámara anterior: Se encuentra entre la córnea y el iris y contiene el humor acuoso. Se podría decir que separa la capa externa anterior de la capa media.
- Cámara posterior: Está situada entre el iris, el cuerpo ciliar y el cristalino, también contiene el humor acuoso.
- Cámara vítrea: Situada entre el cristalino y la retina, contiene el humor vítreo (también conocido como cuerpo vítreo). Estructuralmente, separaría la capa media posterior de la capa interna.

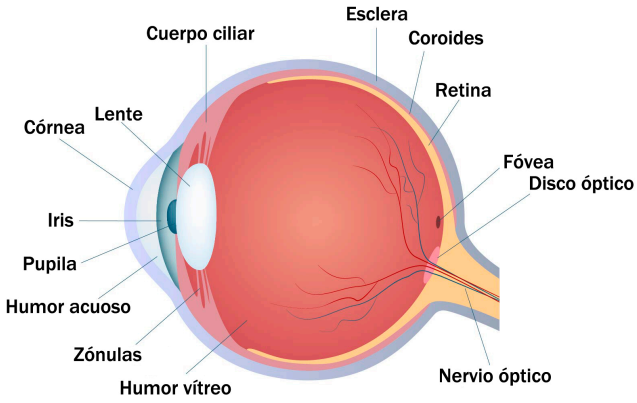


Figura 1. Sección horizontal del ojo.

3.1.3 Estructura óptica

Los fundamentos de formación de la imagen en el ojo, son los mismos que en cualquier sistema óptico. La luz incide en la córnea, refractándose en la misma y en la lente del cristalino para después ser enfocada en la retina. Es en la superficie corneal anterior, cuando la potencia refractiva es mayor, esto se debe a la diferencia que existe entre los índices de refracción del aire (1) y la córnea (1.376). Podemos afirmar por tanto, que los dos elementos en los que se refracta la luz antes de llegar a la retina, son la córnea y el cristalino, en ambos, la luz se refracta en la parte anterior primero y en la posterior después, considerándose la córnea como el elemento más refractivo. Se calcula que la potencia de refracción en la córnea es superior al doble de la del cristalino (40-45D en la córnea, 20D en el cristalino). Una propiedad muy importante del cristalino es su potestad para modificar su potencia en los casos en los que el ojo necesita acomodar a varias distancias, este proceso se conoce como acomodación, esta "modificación" se consigue gracias a la deformación de la lente (cristalino). El iris controla la cantidad de luz que llegará a la retina, siendo la pupila la encargada de regular el diámetro del haz. En un sistema óptico convencional, el iris sería el diafragma y la pupila su abertura.

En los sistemas ópticos, se contabilizan tres pares de puntos cardinales:

- Puntos focales (F y F'): Línea paralela al eje óptico que traza la luz emergente del objeto. Todos los puntos paralelos al eje óptico que proceden de una distancia infinita pasan por el punto focal imagen (F').
- Puntos principales (H y H'): Son conjugados y tienen un aumento lateral de una unidad.
- Puntos nodales (N y N'): También son conjugados sobre el eje, para estos puntos el aumento angular es de una unidad (positiva).

Si considerásemos la aproximación de ojo enfocado al infinito (8.1.1), los puntos se posicionarían como en la figura 2.

Medir la potencia equivalente de cualquier sistema es crucial para determinar ciertas propiedades, en el caso de los sistemas ópticos determina la habilidad para desviar los rayos. La potencia equivalente en un sistema óptico viene denominada por la letra **F** y se define por la siguiente ecuación

$$\mathbf{F} = \frac{n'}{H'F'} = -\frac{n}{HF} \tag{3.1}$$

donde n' es el índice de refracción de la cámara vítrea. Mediante esta ecuación se pueden calcular las distancias focales de un ojo adulto

$$\mathbf{HF = -16.67mm \; y \; H'F' = 22.27mm} \tag{3.2}$$

usando 60D como la potencia equivalente en un ojo adulto y 1.336 para el índice de refracción en la cámara vítrea.

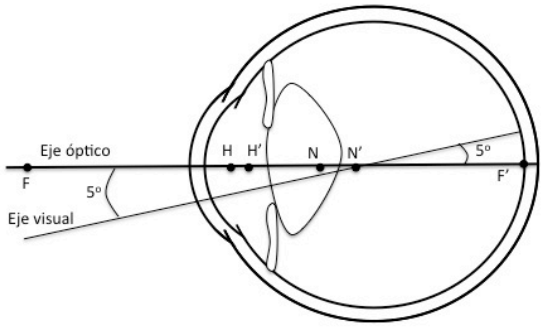


Figura 2. Puntos cardinales y ejes.

3.1.4 Anatomía del globo ocular

En el siguiente punto, trataré los elementos fundamentales del sistema óptico ocular desde el punto de vista anatómico, siendo estos: la córnea, la cámara anterior, el iris y la pupila, el cristalino y la retina.

La córnea se divide en capas y cada una de ellas tiene su propio índice de refracción, la capa más gruesa se denomina estroma (entre 1.36 y 1.38) y su valor está entre el colágeno (1.55) y la sustancia fundamental (1.34). El sistema óptico que forma la córnea separa tres medios distintos, el aire (1), la córnea y el humor acuoso (1.336). De manera general, el valor medio que se le atribuye a la córnea es de 1.376. La "zona óptica" de la córnea es la parte encargada de la formación de la imagen en la retina en visión fotópica, esta zona coincide con la zona centro corneal. En la visión escotópica también interviene la córnea, pero la parte más periférica, este cambio zonal es debido a la dilatación de la pupila.

La cámara anterior contiene el humor acuoso y sirve de separación entre la capa externa (córnea) y la media (iris y cristalino). El humor acuoso está constituido prácticamente por agua (al 98%), gracias a esto se considera un medio homogéneo, de ahí que tenga un índice de refracción totalmente definido.

Como ya he mencionado anteriormente, el iris y la pupila forman el diafragma ocular, siendo el iris el encargado de regular a través de la pupila la cantidad de luz que llegará a la retina. El diámetro de la abertura varía en función de la iluminación, variando de los 2-3mm en visión fotópica hasta los 8mm en visión escotópica. Como muchas partes del cuerpo humano, el ojo también varía con la edad, perdiendo propiedades, adquiriendo rigidez o deteriorándose, en el caso de la pupila, es el tamaño (diámetro) quién decrece a medida que avanzamos en edad. Por ejemplo, en condiciones fotópicas un diámetro típico se consideraría de 4.8mm a los 10 años, a los 45 se situaría en 4mm y a los 80 en 3.4mm.

Situándonos en el marco de un sistema óptico convencional, el cristalino formaría la lente del sistema. El cristalino se encuentra contenido dentro de una cápsula elástica, lo que permite su deformación y favorece por tanto el mecanismo de la acomodación. Para deformar la lente y poder enfocar objetos a varias distancias (figura 1), los ligamentos suspensorios de la zónula de Zinn sostienen la cápsula y controlan la curvatura producida por la tensión de la zónula por acción del músculo ciliar. A mayor tensión en la zónula, mayor "aplastamiento" de la lente, lo que favorece la visión próxima (cristalino en acomodación), a menor tensión, mayor relajamiento de la lente, lo que favorece la visión lejana (cristalino en relajación). Estos cambios en la curvatura de la cápsula, dan lugar a un incremento en la potencia equivalente del ojo. A medida que avanza la edad, la tensión en las zónulas se reduce debido a la rigidez que adquiere el cristalino, esta rigidez viene provocada por la formación de nuevas capas de fibra en las capas externas, aumentando su grosor.

La retina es el elemento más complejo del sistema ocular, esto es debido a que se trata de una prolongación del sistema nervioso central, siendo en este punto donde comienza el proceso de análisis de la información luminosa. La retina está compuesta de dos células fotorreceptoras: los conos y los bastones. Los conos operan en condiciones de luz, por lo que funcionan durante la visión fotópica, los bastones en cambio, lo hacen en condiciones de oscuridad, funcionando en la visión escotópica. En la parte central (mácula lútea), es dónde se concentra una mayor densidad de conos, en el centro de la zona macular se encuentra la fóvea y en el centro de ésta, está la foveóla. La foveóla está compuesta por conos muy finos y es la encargada de la percepción de los detalles. La zona reticular que conecta con el sistema nervioso es conocida como disco óptico o papila óptica. La curiosidad de esta zona, es que constituye un punto ciego, debido a la ausencia de conos y bastones. En un sistema óptico, la retina constituiría la superficie del sensor sobre el que incide la luz proveniente del diafragma.

3.1.5 Ejes del sistema ocular

El hecho de que el ojo no sea simétrico y por tanto, que la fóvea y el punto de fijación no estén situados a lo largo de un eje de simetría adecuado, nos obliga a definir un número determinado de ejes que nos permitan describir las propiedades ópticas del sistema ocular con mayor precisión. Cuando miramos un punto en el espacio, no lo hacemos siguiendo el eje óptico sino a lo largo de una línea que une la fóvea con el punto fijado (punto de fijación), conocemos esta línea como eje visual. El motivo de que no fijemos el punto sobre el eje óptico, es debido que la fóvea no suele estar alineada con la retina (interseccionando el eje óptico). El eje visual es por tanto, el rayo que atraviesa el centro de la pupila y cuyo rayo refractado conjugado incide sobre la fóvea. La línea que une los puntos nodales se conoce como eje nodal, nos permite conocer el tamaño angular de un objeto en el espacio que se encuentra a una determinada distancia.

Aunque los ejes visual y nodal son los más importantes, hay más ejes a tener en cuenta. El eje pupilar es la línea que une el centro pupilar y es normal a la córnea, se utiliza para medir la cantidad de fijación excéntrica. Por último, el eje de fijación es la línea que une el punto de fijación con el centro de rotación. Este eje proporciona la referencia para medir los movimientos oculares.

3.1.6 Campo visual

Rodeando el ojo, hay obstáculos anatómicos que limitan nuestro ángulo de visión monocular, la nariz, la mejilla o la ceja, limitan nuestro campo visual. En el lado nasal por ejemplo, nuestro ángulo de visión es como máximo de 60º, sin embargo, en el lado temporal, dónde no hay obstáculos, se pueden alcanzar los 90º. En el punto 3.1.8 mencioné que el disco óptico constituye un punto ciego en el campo visual monocular debido a la ausencia de células fotorreceptoras, esta "mancha ciega" se sitúa 15º temporalmente y 1.5º hacia abajo con relación al punto de fijación.

Se conoce como campo visual común, la región que se forma al superponerse los campos visuales monoculares. Cualquier punto horizontal al punto de fijación, estaría dentro del campo visual mientras el ángulo descrito entre el eje de fijación y el eje visual no superara los 60º, los puntos situados en la vertical se limitarían de igual manera, 70º en la vertical superior y 90º en la vertical inferior.

3.1.7 Formación de la imagen retiniana

Es muy importante no confundir la imagen retiniana, que es la que se forma en la retina (enfocada o no), con la imagen óptica, que es la imagen completamente enfocada que formaría el sistema refractivo si no existiera la retina.

La imagen que se forma en la retina está invertida horizontal y verticalmente y tiene un tamaño más reducido que el del objeto que la forma, será en el cerebro dónde esta imagen se vuelva a invertir y de este modo adquiera una percepción adecuada.

Para determinar el tamaño y la posición que ocupa la imagen óptica en la retina (aproximándola a una superficie refractiva circular) se puede usar la potencia focal, pero es más común trabajar en términos de vergencia, también expresada en dioptrías. Si la vergencia del objeto y de la imagen se expresan por

$$\text{Vergencia objeto } \boldsymbol{S = \frac{n}{s}} \tag{3.3}$$

$$\text{Vergencia imagen } \boldsymbol{S' = \frac{n'}{s'}} \tag{3.4}$$

dónde n y n' son los índices de refracción en el lado incidente y refractado, respectivamente, s es la distancia objeto y s' la distancia imagen.

Se puede definir el aumento transversal mediante las vergencias con

$$\boldsymbol{B' = \frac{y'}{y} = \frac{n\,s'}{n'\,s} = \frac{S}{S'}} \tag{3.5}$$

Obteniendo el tamaño de la imagen óptica y' mediante:

$$\boldsymbol{y' = y\frac{S}{S'}} \tag{3.6}$$

Para mayor información acerca de las ecuaciones, en el punto 8.1.1 (Anexos) hay un ejemplo de cálculo real usando la aproximación del ojo enfocado a infinito.

3.1.8 Visión binocular

Como ya he definido a lo largo de esta sección, la visión binocular o estereopsis nos permite fusionar la información lumínica obtenida en las dos retinas, para obtener una representación visual en la corteza cerebral. Para poder llevar a cabo esta fusión se han de dar una serie de condiciones:

- El campo visual de un ojo debe superponerse con su homónimo en todas las direcciones posibles, para formar un campo lo suficientemente grande para satisfacer la formación de la imagen retiniana.
- Los campos de fijación monoculares se deben superponer de manera coordinada al movimiento ocular, de modo que los ejes visuales de cada ojo se corten por el mismo punto de fijación.
- La transmisión de la información mediante el nervio óptico (transmisión neuronal), debe estimular la misma área del córtex cerebral para que el cerebro pueda fusionar las dos señales (imágenes retinianas) y formar la imagen final.

3.1.9 Correspondencia

Vamos a considerar la retina como un malla de puntos, al fijar un objeto en el espacio, la imagen de éste se sobreimpresiona en la retina, por lo que cada punto de la imagen pasaría a ocupar una posición en la teórica malla. Si consideramos que el tamaño de las mallas de cada retina es idéntico, la fusión de ambas mallas en una sola, sólo será posible si a cada punto de la malla izquierda le corresponde un punto en la malla derecha. Por lo tanto, dos puntos serán correspondientes si coinciden en dirección y en valor, para ello han de estar proyectados sobre el mismo eje visual e inducir el mismo movimiento motor.

El cálculo de la correspondencia no corresponde a las retinas (ya que carecen de localización espacial), es misión del córtex cerebral procesar los puntos obtenidos en una retina y buscar su

correspondiente en la otra. Si la estimulación cortical se produce en áreas correspondientes, se obtiene una localización direccional, si se estimulan áreas dispares, se obtiene una localización de distancia o profundidad. La localización espacial tiene un carácter subjetivo, por lo que en términos generales todo lo que se percibe "cerca" o "delante de" lleva asociado un movimiento inducido de convergencia (estimulación de las retinas temporales), lo que percibimos "lejos" o "detrás de" lleva asociado un movimiento inducido de divergencia (estimulación de las hemirretinas nasales).

3.1.10 Área de Panum

Para describir el área de Panum, es necesario introducir un par de conceptos básicos, el concepto de diplopía fisiológica y el de horóptero.

La diplopía se conoce como el fenómeno de "visión doble" y no siempre somos capaces de percibirla. Si un objeto está situado próximo o distante al punto de fijación, nuestro cerebro no es capaz de formar una imagen única ya que no encuentra puntos correspondientes en las retinas, por lo que proyecta ambas imágenes. La diplopía fisiológica puede ser de dos tipos, homónima (objetos distantes al punto de fijación) o heterónima (objetos próximos al punto de fijación). La diplopía fisiológica es común al ser humano y la tenemos de manera natural, la diplopía patológica (8.1.2) no, ésta viene asociada al estrabismo.

Se conoce como horóptero, a la región de puntos del campo visual dónde las imágenes estimulan puntos correspondientes. Los objetos que se sitúen por delante o por detrás del horóptero, se verán dobles. Sin embargo, puede ocurrir que puntos cercanos al horóptero (pero no pertenecientes a este) se vean únicos, esto es debido al área de Panum.

Peter Ludvig Panum fue un psicólogo y patólogo danés, nacido el 19 de Diciembre de 1820. En 1858 descubrió que a un punto captado por la retina de un ojo no le correspondía un punto en la retina del otro ojo, sino un área de puntos, esta área de puntos correspondientes es conocida como área de Panum. Todo punto que no esté dentro de esta área, se verá doble.

3.1.11 Proceso de fusión

El proceso de fusión, se produce cuando la transmisión neuronal a través del nervio óptico, estimula el área cerebral necesaria para que se produzca la fusión de las imágenes retinianas. El proceso se debe al casi cruce de los nervios ópticos en el quiasma, esta semi-decusación, provoca que el hemicampo visual derecho se represente en la corteza visual izquierda y el hemicampo izquierdo lo haga sobre la corteza derecha, formando de esta manera la imagen fusionada.

3.2 Fundamentos matemáticos de la visión 3D

3.2.1 Introducción

En este punto, voy a introducir los fundamentos matemáticos de la visión en tres dimensiones desde el punto de vista de los sistemas ópticos no fisiológicos. Ya nos hemos familiarizado con la capacidad que tienen nuestros ojos de percibir el mundo en tres dimensiones, ahora veremos cómo los sistemas computacionales pueden desarrollar estas capacidades. Para generar la imagen 3-D, los sistemas computaciones deben encontrar las correspondencias entre los puntos que se ven en una imagen (imagen izquierda, por ejemplo) y los puntos que se ven en otra imagen (imagen derecha). Con estas correspondencias y conociendo la separación exacta que existe entre las dos cámaras, se puede obtener la localización tridimensional de los puntos. En la práctica el proceso consta de cuatro pasos:

- Eliminación matemática de la distorsión de las lentes. Se suele nombrar este proceso con el término anglófono *undistortion*.
- Ajuste de los ángulos y distancias de cada una de las cámaras. Este proceso es conocido como rectificación.

- Búsqueda de correspondencias entre la imagen izquierda y la imagen derecha. Este proceso recibe el nombre de correspondencia y da como resultado el mapa de disparidad, donde las disparidades son las diferencias en la coordenada x del mismo punto visto desde cada imagen $x^{izquierda} - x^{derecha}$.
- Conociendo la disposición geométrica de las cámaras, podemos convertir el mapa de disparidad en distancias mediante la triangulación. Este proceso se conoce como proyección y da como resultado el mapa de profundidad.

Para situarnos y entender los tres primeros puntos, voy a describir la obtención de la profundidad mediante el proceso de proyección en un caso ideal.

3.2.2 Obtención de la profundidad (caso ideal)

Vamos a asumir que disponemos de un par estéreo perfectamente ajustado, alineado y medido, como el que se observa en la figura 3: dos cámaras cuyos planos imagen son exactamente coplanares el uno del otro, cuyos ejes ópticos son exactamente paralelos y cuyas distancias focales son las mismas ($f^{izq} = f^{der} = f$). También se puede asumir que los puntos principales h^{izq} y h^{der} han sido calibrados para tener las mismas coordenadas respectivas en la imagen izquierda y en la imagen derecha. Es importante no confundir los puntos principales con el centro de la imagen, los puntos principales son aquellos puntos, donde el haz principal corta el plano imagen, esta intersección depende del eje óptico de la lente.

Para avanzar más rápido, asumamos que ambas imágenes están completamente rectificadas, por lo que cada fila de puntos en la imagen izquierda corresponda con la de la imagen derecha (cada punto tenga exactamente la misma coordenada y). Deberíamos asumir también, que somos capaces de encontrar un punto P en la imagen izquierda y derecha (p^{izq} y p^{der}) que tenga respectivamente las mismas coordenadas horizontales, x^{izq} y x^{der}.

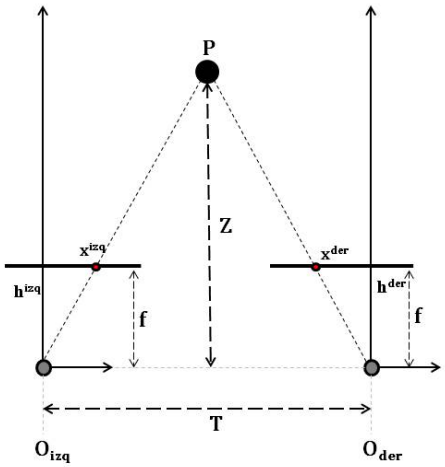


Figura 3. Caso ideal.

En este caso simplificado, podemos definir la disparidad como la diferencia entre las coordenadas horizontales de un punto en ambas imágenes, siendo ésta

$$\boldsymbol{d = x^{izq} - x^{der}} \tag{3.7}$$

Esta situación viene reflejada en la figura 3, dónde es posible deducir la profundidad utilizando triángulos semejantes:

$$\frac{T-(x^{izq}-x^{der})}{Z-f} = \frac{T}{Z} \rightarrow \boldsymbol{Z = \frac{fT}{x^{izq}-x^{der}}} \tag{3.8}$$

Como se observa en la ecuación 3.8, la profundidad es inversamente proporcional a la disparidad, por lo que la relación entre términos no es lineal. Cuando la disparidad es cercana a 0, pequeñas variaciones en la disparidad suponen grandes diferencias en la profundidad, cuando el valor de la disparidad es grande, pequeñas variaciones en la disparidad provocan que la profundidad no cambie demasiado. En la figura 4 se observa con mayor claridad la relación entre la disparidad d y la profundidad Z.

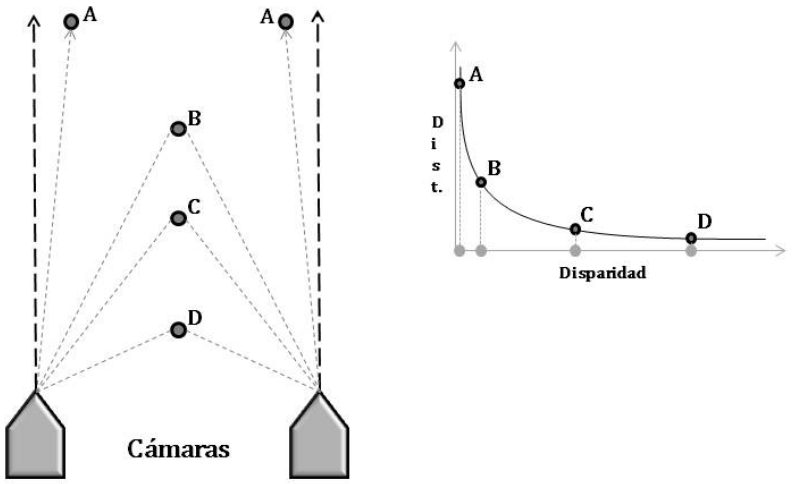


Figura 4. Relación disparidad/profundidad.

Después de la rectificación matemática, las cámaras están alineadas horizontalmente si lo enfocamos desde un punto de vista físico, o verticalmente desde un punto de vista cartesiano (ya que el valor de la coordenada y de dos puntos correspondientes es el mismo), están desplazadas por una distancia T la una de la otra y por la misma distancia focal f (figura 3).

El siguiente paso es aplicar estos conocimientos basados en un entorno ideal, a un entorno real. En la práctica, las cámaras nunca están perfectamente alineadas en la disposición paralela de la figura 3, en su lugar vamos a tener que encontrar matemáticamente las proyecciones de la imagen y los mapas de distorsión que rectificarán las imágenes izquierda y derecha en una disposición paralelo frontal. Es natural remarcar, que cuanto mejor sea el alineamiento y el ajuste físico de las cámaras, más manejable será el matemático, si el ajuste de las cámaras es malo, puede producir imágenes extremadamente distorsionadas. Hay algunas coas, que ni las matemáticas van a poder arreglar.

Otro aspecto que se debe tener en cuenta es la sincronización de las cámaras. Si las cámaras no toman las imágenes exactamente al mismo tiempo, tendremos serios problemas con el movimiento de los objetos en la escena, incluso con el propio movimiento de las cámaras. Si no se sincronizan las cámaras, el uso del sistema se deberá limitar a capturas de escenas de contenido estático con una posición estática del sistema.

En la figura 5 se representa una situación real entre el par estéreo y el ajuste matemático que queremos conseguir. Para realizar correctamente este ajuste, necesitaremos conocimientos en geometría epipolar, una vez asumidos, podremos trabajar de nuevo en el ajuste.

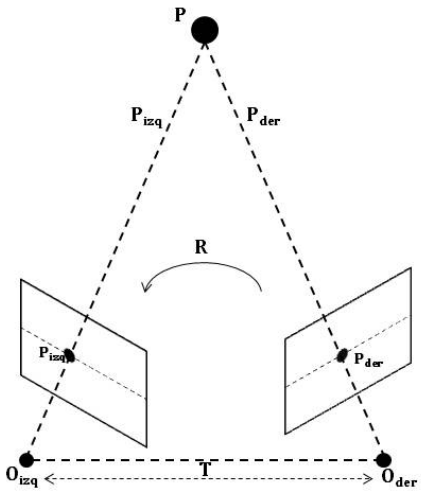


Figura 5. Posible disposición del par estéreo en un entorno real.

Nuestra misión será la de alinear, matemáticamente, el centro de los planos imagen para que los pixeles de las imágenes producidas por las cámaras (izquierda y derecha) estén exactamente alineados (misma coordenada y en los puntos correspondientes).

3.2.3 Geometría epipolar

Como he mencionado al final del punto anterior, será necesario conocer bien los aspectos fundamentales de la geometría epipolar para poder calcular matemáticamente los parámetros necesarios para calibrar el modelo ilustrado en la figura 5. Básicamente, este modelo geométrico combina dos modelos estenopeicos (uno para cada cámara, también conocido como modelo pinhole) con la introducción de dos nuevos puntos, los epipolos. Antes de mencionar para que nos sirve este modelo, vamos a repasar su terminología siguiendo lo descrito en la figura 6.

Para cada cámara existe un centro de proyección separado (O_{izq} y O_{der}) y un par de planos de proyección correspondientes. El punto fijado (P) se proyecta sobre los planos de proyección, intersectando en un punto del plano (p_{izq} y p_{der}). Los epipolos (e_{izq} y e_{der}) son puntos imagen del centro de proyección de la otra cámara (O_{der} respecto de O_{izq}, por ejemplo). El plano físico que se forma entre el punto fijado en la escena (P) y los dos epipolos (e_{izq} y e_{der}) se conoce como plano epipolar y las líneas que unen el centro de proyección del punto fijado (p_{izq} y p_{der}) con los epipolos (e_{izq} y e_{der}) se definen como líneas epipolares.

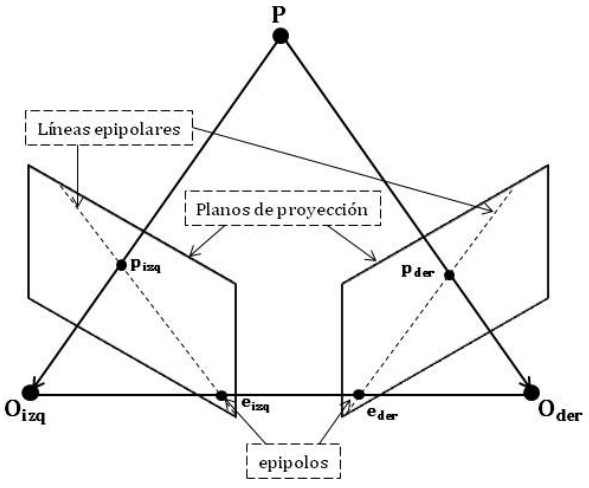


Figura 6. Geometría epipolar.

Cuando se fija un punto y se proyecta sobre los planos de proyección, por ejemplo el izquierdo, el punto puede estar situado a lo largo de la línea de puntos que se forma por el rayo que va

desde O_{izq} y que atraviesa p_{izq} (o O_{der} y p_{der}), desde el punto de vista de la cámara izquierda, no podemos conocer la distancia a la que está el punto que estamos fijando. En este momento, es cuando el epipolo adquiere importancia en el modelo, ya que la imagen de todas las posibles localizaciones del punto fijado, es la línea que pasa a través del punto y el epipolo correspondientes en la otra imagen.

3.2.4 Proceso de rectificación

En el punto 3.2.2, expliqué como se obtenía el mapa de profundidad mediante el proceso de triangulación en un modelo ideal, no obstante, la búsqueda de la correspondencia (3.2.5) en un modelo real, es un proceso más complejo ya que necesitaremos cálculos matemáticos más complicados que los descritos en el apartado.

Como ya mencioné en la introducción (3.2.1), para que el proceso de correspondencia sea lo más fiable posible, es necesario ajustar el sistema al máximo. Las imágenes se pueden rectificar partiendo desde un sistema calibrado, o desde un sistema descalibrado. Para rectificar las imágenes partiendo de un sistema calibrado, podemos usar el algoritmo de Bouguet, si partimos desde un sistema descalibrado, podemos usar el algoritmo de Hartley. Comenzaré definiendo el algoritmo de Bouguet para cámaras calibradas.

El algoritmo de Bouguet se aplica en cámaras calibradas, esto significa que éstas han pasado por el proceso de calibración. Dadas las matrices de rotación y traslación (R, T), el algoritmo de Bouguet trata de reducir los cambios provocados por la proyección en cada una de las imágenes (minimiza las distorsiones provocadas en el proceso de rectificación), al mismo tiempo que maximiza el área de visión común.

Las matrices R y T, se obtienen a partir de la matriz esencial, esta matriz se obtiene en el proceso de calibración y contiene información sobre la translación y rotación de las cámaras en el espacio físico, en este proceso (calibración) se obtiene otra matriz, la fundamental, esta matriz contiene la misma información que la esencial, además de información sobre los parámetros intrínsecos de las dos cámaras.

Continuando con el algoritmo de Bouguet, si queremos minimizar la distorsión en la imagen a causa de la proyección, partiremos de la idea de que la rotación de la matriz R de la imagen derecha es equivalente a dividir su ángulo de giro completo en dos rotaciones r_I y r_D que se aplican a la imagen derecha e izquierda respectivamente. Cada cámara rota hasta la mitad del ángulo de rotación, por lo que sus rayos principales se alinearán de forma paralela. Esta rotación, pone las cámaras en alineación coplanar, pero no alinea sus filas (coordenada y). Para calcular la matriz de rectificación (R_{rect}), que llevará el epipolo de la cámara izquierda a infinito y alineará las líneas epipolares horizontalmente, crearemos una matriz de rotación que estará formada por tres vectores y que comenzará con la dirección de su propio epipolo. Tomando el punto principal (c_x, c_y) dónde comienza la imagen izquierda, el primer vector tendrá la dirección de e_1 y además estará normalizado:

$$\boldsymbol{e_1} = \frac{\boldsymbol{T}}{\|\boldsymbol{T}\|} \tag{3.9}$$

El segundo vector, e_2, tiene que ser ortogonal a e_1, por lo que una solución interesante serñia escoger un vector ortonormal al rayo principal de la cámara izquierda:

$$\boldsymbol{e_2} = \frac{[-\boldsymbol{T_y}\boldsymbol{T_x}\boldsymbol{0}]^T}{\sqrt{\boldsymbol{T_x^2}\boldsymbol{T_y^2}}} \tag{3.10}$$

El tercer vector tiene que ser ortogonal a e_1 y e_2:

$$\boldsymbol{e_3} = \boldsymbol{e_1} \times \boldsymbol{e_2} \tag{3.11}$$

La matriz R_{rect} queda de la siguiente manera:

$$\boldsymbol{R_{rect}} = \begin{bmatrix} (\boldsymbol{e_1})^T \\ (\boldsymbol{e_2})^T \\ (\boldsymbol{e_3})^T \end{bmatrix} \tag{3.12}$$

Como la idea principal es conseguir un alineamiento vertical entre las dos cámaras, se aplica lo siguiente:

$$\boldsymbol{R_{izq}} = \boldsymbol{R_{rect}}\boldsymbol{r_1} \tag{3.13}$$

$$\boldsymbol{R_{der}} = \boldsymbol{R_{rect}}\boldsymbol{r_D} \tag{3.14}$$

Este método también calcula las matrices de proyección (P_{D_rect} y P_{I_rect}):

$$\boldsymbol{P_{I\_rect}} = \boldsymbol{K_{rect\_I}}\boldsymbol{P'_I} = \begin{bmatrix} \boldsymbol{f_{x\_I}} & \boldsymbol{s_I} & \boldsymbol{c_{x\_I}} \\ \boldsymbol{0} & \boldsymbol{f_{y\_I}} & \boldsymbol{c_{y\_I}} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{1} \end{bmatrix} \begin{bmatrix} \boldsymbol{1} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{1} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{1} & \boldsymbol{0} \end{bmatrix} \tag{3.15}$$

$$\boldsymbol{P_{D\_rect}} = \boldsymbol{K_{rect\_D}}\boldsymbol{P'_D} = \begin{bmatrix} \boldsymbol{f_{x\_D}} & \boldsymbol{s_D} & \boldsymbol{c_{x\_D}} \\ \boldsymbol{0} & \boldsymbol{f_{y\_D}} & \boldsymbol{c_{y\_D}} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{1} \end{bmatrix} \begin{bmatrix} \boldsymbol{1} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T_x} \\ \boldsymbol{0} & \boldsymbol{1} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{1} & \boldsymbol{0} \end{bmatrix} \tag{3.16}$$

s_I y s_D corresponden al factor de distorsión por pixel, que en las cámaras modernas casi siempre es 0.

Las matrices de proyección llevan un punto 3D en coordenadas homogéneas a un punto 2D, de la siguiente manera:

$$\boldsymbol{P}\begin{bmatrix} \boldsymbol{X} \\ \boldsymbol{Y} \\ \boldsymbol{Z} \\ \boldsymbol{1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ \boldsymbol{w} \end{bmatrix} \tag{3.17}$$

donde las coordenadas pueden ser calculadas como (x/w, y/w). Los puntos 2D en el espacio se pueden proyectar en puntos 3D dadas las coordenadas y las parámetros intrínsecos de la cámara. La matriz de proyección será:

$$\boldsymbol{Q} = \begin{bmatrix} \boldsymbol{1} & \boldsymbol{1} & \boldsymbol{0} & \boldsymbol{-c_x} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{-c_y} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{f} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{-1/T_x} & \boldsymbol{(c_x - c'_x)/T_x} \end{bmatrix} \tag{3.18}$$

En esta ecuación los parámetros son los de la imagen izquierda a excepción de c_x que es el la coordenada x del punto principal en la imagen derecha. Si el rayo principal corta en infinito, entonces $c_x = c'_x$ por lo que $(c_x - c'_x)/T_x$ será nulo. Dado un punto 2D de coordenadas homogéneas y su disparidad d, podemos proyectar el punto en tres dimensiones utilizando:

$$\boldsymbol{Q}\begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ \boldsymbol{d} \\ \boldsymbol{1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{X} \\ \boldsymbol{Y} \\ \boldsymbol{Z} \\ \boldsymbol{W} \end{bmatrix} \tag{3.19}$$

Las coordenadas tridimensionales serán entonces (X/W, Y/W, Z/W).

Visto el algoritmo de Bouguet, que se aplica para sistemas calibrados, veamos ahora el algoritmo de Hartley, que se aplica para sistemas descalibrados.

El algoritmo de Hartley trata de encontrar homografías que aproximen los epipolos a infinito, reduciendo de esta manera las diferencias entre el par de imágenes.

La ventaja de este algoritmo, es que la calibración estéreo puede llevarse a cabo simplemente mediante la observación de puntos en la escena, siendo muy interesante para calibración de contenido dinámico (streaming). La desventaja, es que no tendremos conocimiento de la escala de la imagen que estemos procesando. Por ejemplo, si usáramos un tablero de ajedrez para encontrar las coincidencias, no podríamos verificar si el tablero mide 100 metros o 100 centímetros o si está cerca o lejos de la cámara. Tampoco conocemos los parámetros intrínsecos de la cámara, por lo que las distancias focales, desvío en los píxeles, centros ópticos y puntos principales son los mismos en las dos imágenes. Por lo tanto, solo podemos determinar la reconstrucción tridimensional de un objeto desde una transformación proyectiva. Esto significa que las diferentes escalas o proyecciones de un objeto puedan parecernos iguales, lo que es un serie problema si lo que queremos es conocer la profundidad de los puntos.

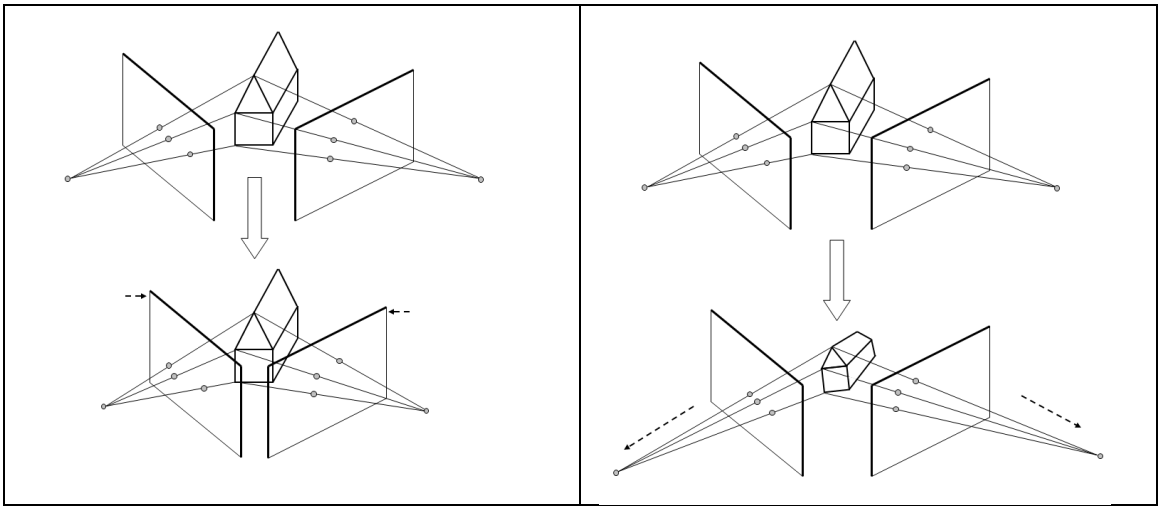


Figura 7. Ambigüedad en la reconstrucción (Hartley).

Es importante mencionar que aunque este algoritmo es capaz de calcular la matriz fundamental sin que haya habido un proceso previo de calibración, no es demasiado recomendable, ya que una no calibración de las cámaras estéreo puede conllevar a imágenes distorsionadas, donde el resultado tras aplicar el algoritmo de Hartley, sería nefasto.

De este punto podemos desprender, que aunque el algoritmo de Hartley pueda rectificar las imágenes sin una calibración previa, el resultado aplicando el algoritmo a un sistema calibrado, siempre será mejor. Así que siempre que se pueda, se deberá calibrar el sistema para obtener los parámetros intrínsecos y extrínsecos de las cámaras.

3.2.5 Proceso de correspondencia

Definimos el proceso de correspondencia como la búsqueda de un punto tridimensional a partir de dos imágenes (la vista de dos cámaras). El cálculo de la correspondencia, sólo es posible en las áreas visibles dónde los campos visuales de las cámaras se solapan. Una vez más, vuelvo a recalcar la importancia de la calibración y la rectificación para obtener un mejor resultado en este proceso. Por lo tanto, una vez obtenemos las coordenadas físicas de las cámaras o los tamaños de los objetos de la escena, se pueden deducir las medidas de la profundidad a partir de las medidas de la disparidad obtenidas mediante la triangulación $d = x^{izq} - x^{der}$ (o $d = x^{izq} - x^{der} - (c_x^{izq} - c_x^{der})$, si el rayo principal corta en infinito) entre los puntos correspondientes de las dos imágenes.

Hay una gran cantidad de algoritmos para calcular la correspondencia, como sería muy extenso definirlos todos, me voy a centrar en los utilizados en la aplicación: *block-matching* (BM) y *Semiglobal Matching* (SGM).

El algoritmo por búsqueda de correspondencias estéreo (block-matching) propuesto por Kurt Konolige (7.2 [2]), calcula sus resultados utilizando ventanas que computan las coincidencias

por medio de suma de diferencias absolutas (ventanas SAD) entre píxeles de la imagen izquierda y derecha, para ello se desplaza la ventana una cantidad determinada de pixeles sobre la imagen izquierda respecto de la derecha. Este algoritmo, sólo encuentra puntos "altamente coincidentes" entre las dos imágenes. Para entenderlo de manera gráfica, en una escena de mucha textura como un paisaje, es más fácil detectar la profundidad de cada píxel, en una escena de poca textura, como el pasillo de una casa, se puede detectar la profundidad en muy pocos puntos. A fin de mejorar la calidad del mapa de disparidad resultante, este algoritmo incluye una etapa de prefiltrado previa al cálculo de la correspondencia y una etapa de postfiltrado posterior. Por lo tanto, se distinguen tres fases:

1. Pre-filtrado para normalizar el brillo de la imagen y mejorar su textura.
2. Búsqueda de la correspondencia a lo largo de las líneas epipolares, utilizando la ventana SAD.
3. Post-filtrado para eliminar la correspondencia en malas coincidencias.

Debido a la existencia de dos lentes en las cámaras estéreo, se producen unas diferencias de iluminación y de perspectiva entre las imágenes izquierda y derecha, la etapa de prefiltrado busca disminuir el efecto de esta diferencia. Para ello se emplea el método de normalización de intensidades. Cada intensidad dentro de un área correlacionada, se normaliza por el promedio de las intensidades de esa área. Por lo tanto, el píxel central de la ventana I_c se reemplaza por:

$$\mathbf{min}(\mathbf{max}(\boldsymbol{I_c - T}, -\boldsymbol{I_{lim}}), \boldsymbol{I_{lim}}) \tag{3.22}$$

Siendo T el valor promedio de intensidad en la ventana e I_{lim} el límite numérico positivo.

De la primera etapa se obtiene la diferencia absoluta L_1 que se usará para calcular la correlación entre las dos ventanas. Para la optimización de los cálculos, se fuerza a que el tamaño de la ventana sea *n x n* y que el valor máximo de la disparidad sea múltiplo de 16. Por tanto, se define la correlación como:

$$\boldsymbol{C_{BM}(x^l, y^l, d)} = \frac{1}{n^2}\sum_{i=x^l-\frac{n}{2}}^{x^l+\frac{n}{2}}\sum_{j=y^l-\frac{n}{2}}^{y^l+\frac{n}{2}}\left|\boldsymbol{I_{izq}(i,j) - I_{der}(i-d,j)}\right| \tag{3.23}$$

Y el mapa de disparidad resultante:

$$\boldsymbol{D_{BM}(x^l, y^l)} = \boldsymbol{arg}\,\mathbf{min}_{\boldsymbol{d}\in[\mathbf{0},\boldsymbol{D}]}\,\boldsymbol{C_{BM}(x^l, y^l, d)} \tag{3.24}$$

Una vez se calcula la correspondencia, se aplica la etapa de postfiltrado. Esta etapa se utiliza para eliminar las zonas con poca textura y los errores causados en las zonas ocluidas de las dos imágenes.

El uso de este algoritmo está muy extendido globalmente, encontrándose en diversos estudios y software específico. Tanto es así, que se encuentra implementado en la librería de visión artificial **OpenCV**, la cual se ha utilizado en la aplicación desarrollada.

El segundo algoritmo, *Semiglobal Matching* (SGM), que fue propuesto por H. Hirschmüller (7.2 [3]), se basa en la idea de hacer coincidir los píxeles de las imágenes estéreo, haciendo múltiples aproximaciones unidimensionales, logrando de esta manera una aproximación global en dos dimensiones. Para lograr esto, el autor propone la ecuación de energía:

$$\boldsymbol{E(D)} = \sum_{\boldsymbol{P}}\begin{Bmatrix}\boldsymbol{C(p, D_p)} + \sum_{\boldsymbol{q}\in\boldsymbol{N_p}}\boldsymbol{P_1 T}[|\boldsymbol{D_p - D_q}| = \mathbf{1}] \\ + \sum_{\boldsymbol{q}\in\boldsymbol{N_p}}\boldsymbol{P_2 T}[|\boldsymbol{D_p - D_q}| > \mathbf{1}]\end{Bmatrix} \tag{3.25}$$

El primer término se corresponde con la suma de todo pixel coincidente para la disparidad D. El segundo término suma una constante P_1 para todos los píxeles (q) vecinos de p (N_p sería el "vecindario" de p), para el cual, los cambios de disparidad son mínimos (1 píxel). El tercer término suma una constante de penalización P_2, que penaliza cambios de disparidad más

bruscos. Utilizando una penalización baja para pequeños cambios se favorecen las superficies curvas, la penalización para cambios grandes preserva las discontinuidades. Estas discontinuidades, son perceptibles como cambios en la intensidad. Esto se logra adaptando el valor de P_2 a la intensidad del gradiente, siendo:

$$\boldsymbol{P_2 = \frac{P_2'}{|I_{bq} - I_{bp}|}} \tag{3.26}$$

Este valor correspondería al vecindario de los píxeles p y q situados en la imagen base (I_b). Siempre se debe asegurar que P_2 sea mayor que P_1 ($P_2 > P_1$).

Se podría formular el problema de la correspondencia estéreo, como la búsqueda de la disparidad D que minimiza la función de energía E(D) (3.25). Debido a que se trabaja en dos dimensiones, N_p presenta un problema para muchas discontinuidades, para facilitar el cálculo, el autor propone añadir múltiples costes de correspondencia en un espacio unidimensional. Los costes añadidos por la suavidad *S (p,d)* para un pixel p y una disparidad d, se calculan sumando el coste de las diferentes rutas que terminan en el pixel p con la disparidad d. El coste a través de la ruta de un pixel p a una disparidad d con dirección r se define como:

$$\boldsymbol{L_r'(p,d) = C(p,d) + \min\begin{Bmatrix} L_r'(p-r,d), L_r'(p-r,d-1) + P_1, \\ L_r'(p-r,d+1) + P_1, \\ \min_i\{L_r'(p-r,i) + P_2\} \end{Bmatrix}} \tag{3.27}$$

El autor coincide en que el número de rutas óptimo para calcular la suavidad es de 16, situando la discusión en el estudio realizado en 7.3[3], evaluando desde 8 hasta 16 rutas. Quedando la suavidad:

$$\boldsymbol{S(p,d) = \sum_r L_r(p,d)} \tag{3.28}$$

3.3 Introducción a stereoTuner

El objetivo primordial de la aplicación es elaborar un mapa de disparidad lo más adecuado posible, partiendo de un par de imágenes tomadas por una cámara estéreo. La aplicación dispone de dos algoritmos para realizar el calibrado: stereoBM y stereoSGBM.

Ambos algoritmos se utilizan para el procesado de la correspondencia en imágenes estéreo, stereoBM y stereoSGBM son los nombres que reciben las clases de **OpenCV**. stereoBM obtiene como resultado el mapa de disparidad utilizando el algoritmo "block matching", mientras que stereoSGBM lo hace utilizando "Semiglobal matching" (algoritmo propuesto por H. Hirschmüller, 2008). El usuario de stereoTuner podrá parametrizar los algoritmos mediante los ajustes del panel de calibración, para así poder conseguir un mejor resultado. En el punto 3.4 veremos con mayor detalle los aspectos estructurales de la aplicación.

stereoTuner se configura y se inicia a partir de un perfil de configuración, los perfiles son unos archivos **xml** con una extensión propia de la aplicación (.st) y unos *tags* propios definidos y procesados por un *parser* desarrollado para la aplicación. Para cargar el perfil de configuración, la aplicación accede a un **xml** de configuración llamado stereoTuner-config.xml. Desde este archivo se busca el perfil y se cargan las preferencias: propiedades, imágenes estéreo, mapas de color y ficheros de calibración. La primera vez que se inicia stereoTuner, el fichero no tiene informado un perfil de configuración, por lo que el *XMLsT-parser* no encuentra un valor y el programa detecta que no puede configurar las preferencias, entonces, se cargan unas preferencias por defecto y se abre el asistente de perfiles (3.4.12), en este punto, la aplicación no dejará salir del asistente hasta que no hayamos creado un nuevo perfil o hayamos cargado uno existente mediante el selector de archivos (3.4.10).

Los ficheros de calibración, son los ficheros en los que se guarda la información respectiva al panel de calibración. Actualmente, hay dos tipos de ficheros, los correspondientes al algoritmo

stereoBM, cuya extensión es BM, y los correspondientes al algoritmo stereoSGBM, cuya extensión es SGBM. Opté por separar los datos de calibración del perfil de configuración, por el mero hecho de dar más libertad al usuario y que desde una misma sesión (trabajando con un solo perfil), pudiera probar y crear varias calibraciones, sin tener que crear un perfil distinto para cada una.

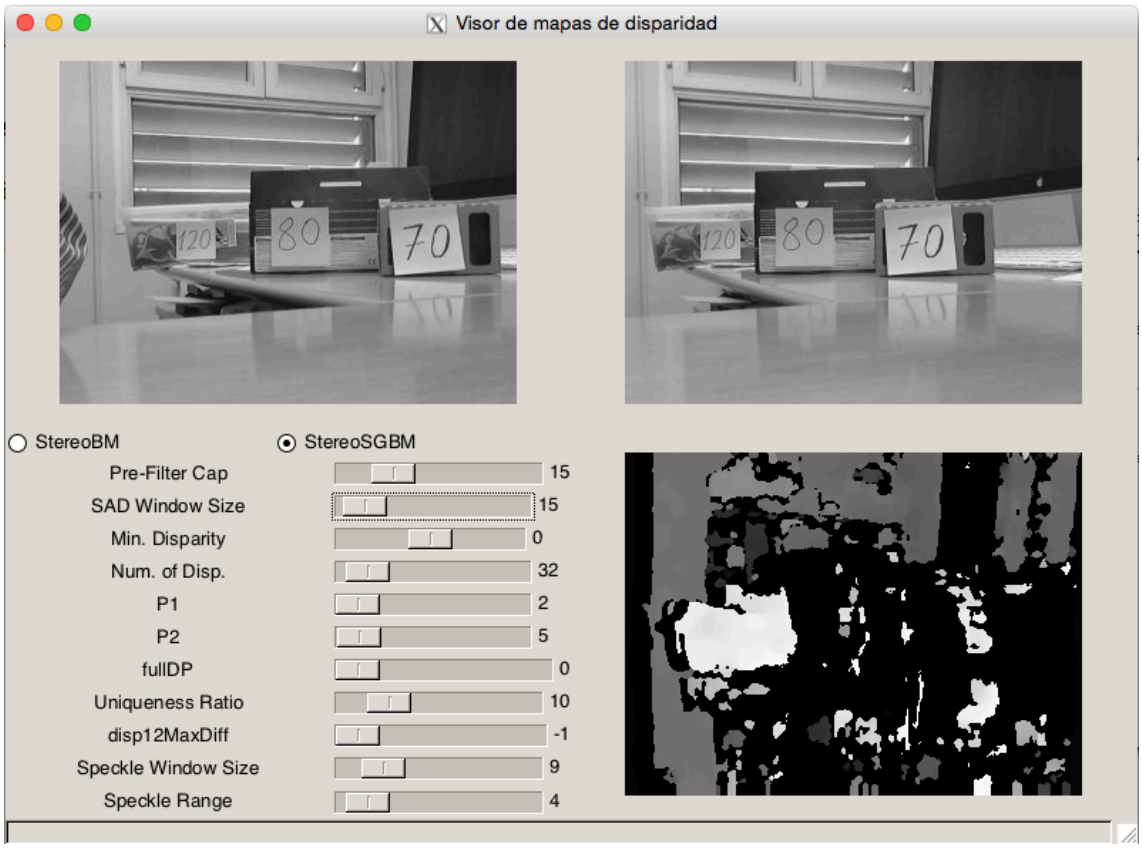


Figura 8. Esqueleto de partida.

Otro punto a tener en cuenta es la herramienta "Capture", a la cual se accede mediante la barra de herramientas. La herramienta "Capture" nos permite fijar un punto en la imagen izquierda, mostrándonos la disparidad exacta en el área de información y utilizando el cursor del ratón para movernos por la imagen derecha, obtener la diferencia de coordenadas con respecto al punto fijado. Esto permite comprobar personalmente cual es la disparidad, podríamos decir, que es la manera de comprobar mediante un "humano" la disparidad en el punto fijado, a fin de comprobar si la disparidad obtenida mediante el algoritmo seleccionado, es la correcta.

stereoTuner dispone de más funcionalidades que se detallarán en el punto 3.6, como por ejemplo: la posibilidad de pintar líneas horizontales en las dos imágenes para comprobar si la imagen está bien rectificada, poder sobreimpresionar mediante diversas escalas de colores el mapa de disparidad, en función de la disparidad o la profundidad, disponer de un sistema cronológico de backup de acciones, que el usuario puede exportar en cualquier momento, la posibilidad de guardar ajustes desde la ventana de preferencias, elegir en el área de información entre los valores de disparidad o profundidad, etc.

No cabe duda que stereoTuner es una aplicación en expansión y que dispone de muchas líneas de actuación futuras, algunas de ellas las nombraré en el cuarto capítulo de la memoria.

3.4 Estructura de la aplicación

3.4.1 Pantalla principal

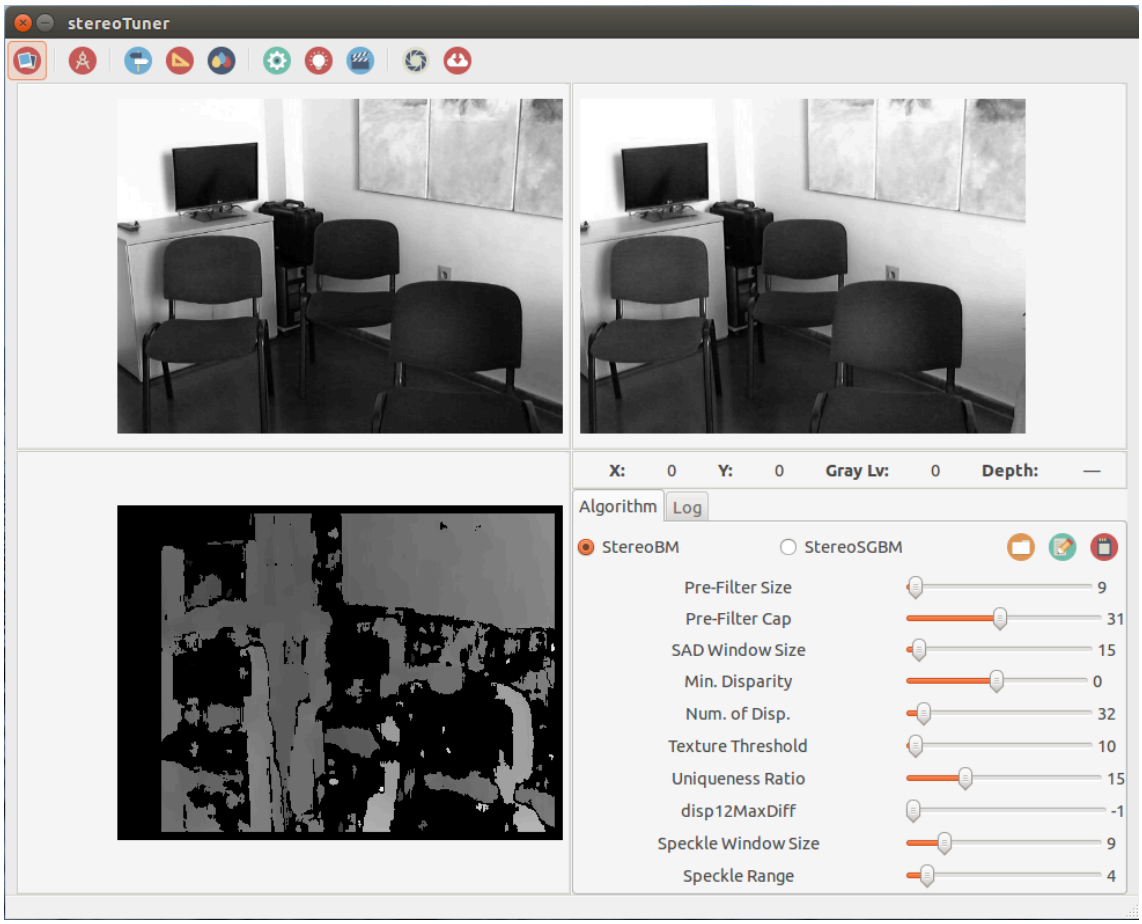


Figura 9. Pantalla principal de la aplicación.

Cuando arrancamos la aplicación, la primera ventana en aparecer es la ventana principal, desde la cual se llevan a cabo la mayoría de las funciones de stereoTuner. En la figura 9, se pueden observar los componentes principales que la forman: en la parte superior se encuentra el *toolbar*, o barra de herramientas; inmediatamente debajo, se encuentra el par de imágenes estéreo; en la parte inferior, a la izquierda, se encuentra el mapa de disparidad y a la derecha, se encuentra el *notebook*, que engloba dos vistas que se navegan mediante un sistema de pestañas; la vista *Algorithm*, dispone de todos los parámetros modificables de cada algoritmo y la vista *Log*, lista cronológicamente todas las acciones que el usuario y/o stereoTuner van realizando.

El *logger* nos ofrece distintas posibilidades dependiendo del rol que desempeñemos a la hora de iniciar stereoTuner, si lo hacemos como usuario, el *logger* nos listará todas las acciones que vamos realizando a modo de auditor de actuaciones, si en cambio estamos desarrollando en stereoTuner, lo podremos utilizar como un inspector, ya sea para pintar avisos, valores de variables, o lo que consideremos oportuno. En el punto 3.6.5 repasaré los distintos tipos de *logs*, su estructura y detallaré el patrón de diseño que se ha seguido para implementarlo.

3.4.2 Toolbar

El *Toolbar* de stereoTuner es el sitio desde el cual se realizan o inician, la mayoría de las acciones de la aplicación. Esta barra es realmente una botonera, dentro de la cual se distinguen tres tipos de elementos: botones, botones tipo *toggle* y separadores. Los botones, lanzan una acción al ser pulsados, por lo tanto, no tienen estado. Cuando se pulsa un botón en stereoTuner se realiza una acción inmediata, normalmente el acceso a un método concreto. Los botones de tipo *toggle* o *toggle buttons*, tienen una diferencia clave (y primordial) respecto a los botones

convencionales, tienen estado, esto quiere decir, que un *toggle button* puede estar activo o inactivo. Esta funcionalidad se usa principalmente para las acciones no desencadenantes, es decir, las que se ejecutan (o no) de forma continuada. Por último, los separadores tienen una funcionalidad sin un mayor impacto que el visual, se usan para separar agrupaciones de botones, generalmente separan grupos de botones que realizan acciones muy dispares.

Figura 10. Barra de herramientas.

Si observamos la figura 10, la barra de herramientas consta de 10 botones, agrupados en 5 grupos. El primer botón, abre la ventana del asistente de perfiles de configuración (3.4.12). El segundo botón activa la propiedad de rectificación, esta propiedad no está totalmente implementada en el código, pero si está iniciada y detallada para su desarrollo, ya que como he comentado anteriormente, stereoTuner es una aplicación en expansión, que de seguro continuará creciendo, esta funcionalidad es por lo tanto, una de las líneas futuras que se comentarán más adelante.

Los siguientes tres botones, que forman parte del tercer grupo, son los botones que actúan sobre la pantalla principal. El primero alterna entre la obtención de los valores de disparidad o de la profundidad (estos valores se pintan en el área de información, punto 3.4.4). El segundo implementa la funcionalidad de las líneas de rectificación, función que veremos en detalle en el punto 3.6.6. El tercero y último, alterna la escala de color del mapa de disparidad, si el botón está inactivo, el mapa de disparidad se coloreará en escala de grises, si está activo, se coloreará en función de la escala seleccionada (puntos 3.4.9 y 3.4.11).

El siguiente grupo de botones, son botones que como el primero, redirigen a otra ventana distinta a la principal. El primero abre la ventana de preferencias (3.4.9), ventana desde la cual, podremos modificar todos los parámetros de la aplicación, así como acceder a otras ventanas, como el selector de archivos (3.4.10) o el selector de mapas de color (3.4.11). El segundo botón abre una ventana con algunos consejos rápidos acerca de la aplicación, como por ejemplo, la secuencia de acciones que hay que llevar a cabo en la herramienta "Capture". El tercer botón (y último de este cuarto grupo) abre una ventana que reproduce un vídeo explicativo con las funciones de stereoTuner y la manera de llevarlas a cabo.

El último grupo son botones que actúan sobre bloques de la pantalla principal. El primer botón es el correspondiente a la herramienta "Capture", como en este caso la secuencia de acciones para llevar a cabo la funcionalidad completa de la herramienta, es algo más compleja que pulsar el botón, la detallaré más adelante en el punto 3.6.12. El segundo y último botón del grupo, es el correspondiente a la ventana de exportación del *logger* (3.4.15), en esta ventana podremos exportar la información del *logger* en un fichero de texto, con el nombre que previamente hemos introducido por teclado.

3.4.3 Imágenes estéreo

Inmediatamente después del *toolbar* (fig. 10), encontramos las imágenes de la cámara estéreo. Cada imagen se encuentra dentro de un panel que las posiciona en función de su tamaño. Dentro del panel existe una jerarquía de objetos **GTK** que realizan distintas funciones, conceptualmente funcionan como una matrioska, hay un objeto top que encapsula otro objeto (*child*), que a su vez encapsula otro objeto, así hasta la imagen que se visualiza.

En la parte exterior está el *frame* que es quién delimita el panel y obtiene los estilos (color de fondo, borde…), dentro del *frame* se encuentra el *alignment*, que es quién posiciona el siguiente objeto, el *eventbox* o caja de eventos (es quien engloba a la imagen), el alineamiento en el eje x para el panel izquierdo, es al final con un margen de 10 px (para que no coincida con el borde del panel) y en el panel derecho, al principio con el mismo margen. En el caso del eje y, el alineamiento es al 50% (centrado), común para ambos paneles. Por lo tanto, dentro del alineamiento se encuentra la caja de estados, que es quién engloba a la imagen. La caja de estados, es posiblemente una de las partes estructurales más importantes de ambos paneles, ya

que están configuradas para realizar una acción en respuesta a un evento definido, por ejemplo un clic (usado en la herramienta "Capture"), o el desplazamiento del ratón, usado para obtener las coordenadas x e y, el nivel de gris y la disparidad en el píxel exacto en el que se encuentre el puntero. Por último, dentro del *eventbox* se encuentra la imagen que se visualiza.

3.4.4 Área de información

El área de información, se encuentra posicionada en la parte superior del mismo panel que el *notebook*, aunque son bloques totalmente independientes, coexisten dentro del mismo panel. El bloque tiene 4 elementos, el valor de la coordenada x, el valor de la coordenada y, el nivel de gris y el valor de la disparidad/profundidad (en función del estado del tercer botón del *toolbar*). Este bloque está en constante cambio y es posiblemente el panel que más acciones recibe de stereoTuner, ya que siempre que el ratón está encima de cualquiera de las imágenes estéreo o el mapa de disparidad, actualiza sus valores por los respectivos del píxel actual. Además, este panel modifica sus literales en dos puntos: al clicar el botón selector de disparidad/profundidad y al accionar la herramienta "Capture". Al clicar el botón selector de disparidad/profundidad, si el estado pasa a activo, el cuarto literal del bloque muestra "Disparity" y su valor es la disparidad del pixel actual en la imagen de disparidad, si el botón pasa a inactivo, se muestra "Depth" y su valor es el de la profundidad. En el caso de la herramienta "Capture", los dos primeros literales (x e y), se modifican por: "Lpx" y "Rpx", el literal correspondiente al nivel de gris, se convierte en el literal "Diff" y el literal disparidad/profundidad se pone en modo disparidad (independientemente del modo en el que esté antes de pulsar el botón). El valor de "Lpx", muestra las coordenadas x e y de manera continua hasta que se clica sobre la imagen izquierda, en ese momento el valor se queda fijo, al igual que el valor de la disparidad. El valor de "Rpx" muestra el valor de las coordenadas del punto sobre el que estamos al movernos sobre la imagen derecha y "Diff" muestra la diferencia de coordenadas entre el punto fijado y el punto actual (en la imagen derecha).

3.4.5 Mapa de disparidad

En la parte inferior de la pantalla principal, en el panel situado a la izquierda, se encuentra el mapa de disparidad. La estructura del panel es idéntica al de la imagen izquierda. En el tope jerárquico estaría el *frame*, después el *alignment* (idénticas propiedades que el de la imagen izquierda), después el *eventbox* y al final la imagen. Este panel modifica levemente su aspecto en función del estado del botón de selección de color (5ª posición en el *toolbar*), si está activo, el mapa de disparidad aparecerá coloreado siguiendo la escala seleccionada en el menú de preferencias (3.4.9), si está inactivo, aparecerá en escala de grises.

Como dato curioso, decidí seguir mostrando el valor de gris en el área de información, aún a pesar de que el mapa estuviese coloreado con una escala seleccionada, ya que mostrar el valor de esa escala no tendría ningún uso útil.

3.4.6 Notebook – Algorithm

A la derecha del panel del mapa de disparidad, se encuentra el notebook. Este bloque muestra una ventana distinta para cada pestaña, la opción por defecto es la pestaña "Algorithm", que se divide en dos partes. En la parte superior de la ventana, se encuentran los algoritmos a seleccionar por el usuario (actualmente hay dos *radio buttons*, uno para stereoBM y otro para stereoSGBM) y un grupo de tres botones. Los *radio buttons*, tienen la peculiaridad de formar agrupaciones de botones, es decir, si A está seleccionado no lo podrá estar B, y viceversa. Opté por los *radio buttons* por una sencilla razón, sólo implementé dos algoritmos, aunque al desarrollar esta parte, ya contemplé la posibilidad de que se usaran más algoritmos, por lo que desarrollé el código para sustituir los botones por un combo selector (fig. 11), para si en un futuro se ampliaban los algoritmos, el desarrollador encargado sólo tuviera que descomentar las líneas en el código.

Figura 11. Combo selector.

Los tres botones, atañen a las funcionalidades del panel de calibración. El primer botón, abre un selector de archivos que permite cargar ficheros de calibración (BM o SGBM), el segundo botón abre la ventana de creación de ficheros de calibración (3.4.8) y el tercero guarda los cambios realizados en el fichero de calibración en el que se esté trabajando. Si se clica el botón de guardado, con el *radio button* del algoritmo StereoBM seleccionado, se modificará el fichero de calibración actual de extensión BM, si es el *radio button* del algoritmo StereoSGBM el seleccionado, se modificará el fichero con extensión SGBM.

La segunda parte de la ventana, es el menú de calibración, este bloque está compuesto de una serie de barras de ajuste que actúan sobre los parámetros modificables de cada algoritmo. Es importante recalcar que dependiendo del algoritmo que se seleccione, este bloque cambiará, ya que los dos algoritmos no comparten los mismos parámetros, ni en número ni en forma.

Las barras de ajuste están conectadas mediante una señal de "change-value", es decir, cada vez que modificamos el valor de una barra, se invoca un método que cambia el valor que a su vez, invoca al método del cálculo del mapa de disparidad, provocando un cambio instantáneo en la imagen del panel del mapa de disparidad.

3.4.7 Notebook – Log

En la otra pestaña del *notebook*, se encuentra la ventana del *logger* (fig. 12). Visualmente, la ventana está compuesta por una tabla de 2 columnas, la primera informa el tiempo en horas y minutos y la segunda informa el mensaje que se ha producido en ese instante. Cada acción que se va realizando, se va añadiendo a la lista, por lo que la lista mantiene un orden cronológico.

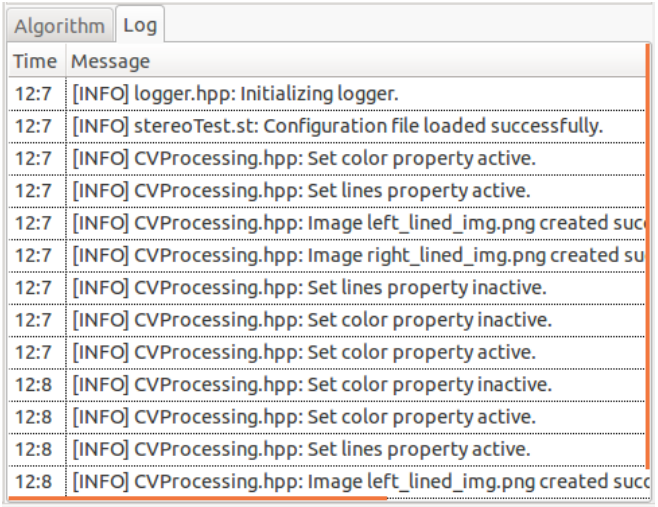


Figura 12. Logger.

Estructuralmente, esta ventana está compuesta por varios objetos: un *TreeView*, un *ListStore*, un *TreeIter* y un *CellRenderer*. El funcionamiento es el siguiente: el *TreeView* es la vista sobre la que se apoya la lista (*ListStore)*, cada vez que se añade una fila nueva, se crea dicha fila (vacía) en la lista y mediante el *CellRenderer* se renderiza la fila con el valor adecuado. Para que la vista (*TreeView)* sepa en qué posición se ha de colocar la nueva fila, se usa un iterador (*TreeIter*), que es algo similar a una variable de posición.

3.4.8 Ventana de creación de ficheros de calibración

En stereoTuner, las ventanas (a excepción de la principal) tienen una estructura común a modo de panel, se componen de una cabecera y un cuerpo. En la parte superior se ubica la cabecera, que contiene el título de la ventana y está coloreada en el naranja oficial de Ubuntu (#52E920). El resto de la ventana, lo compondría el cuerpo y es dónde se concentra toda la estructura propia de la ventana.

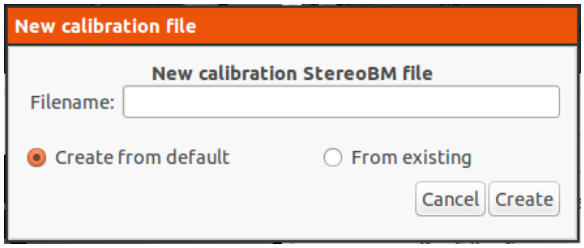


Figura 13. Ventana de creación de ficheros de calibración.

Al clicar el segundo botón del panel de calibración (3.4.6), se abre una ventana que nos permite crear ficheros de calibración. El cuerpo de la ventana está compuesto por un título, que varía en función del algoritmo que esté seleccionado en el momento del acceso. Si clicamos en el botón, cuando esté seleccionado el algoritmo stereoBM, el título mostrará el literal que se ve en la figura 13, si el algoritmo seleccionado es stereoSGBM, mostrará: "New calibration StereoSGBM file". Inmediatamente después del título, se encuentra la entrada de texto para recoger el nombre del fichero. Debajo de la entrada, se encuentra el literal de error, este mostrará en rojo el resultado de la validación en caso de que se introduzcan caracteres no imprimibles. Debajo del literal de error, se observan dos *radio buttons*, el primero indica que el fichero de calibración a generar, partirá del contenido por defecto (src/config/default.BM) y el segundo indica que el fichero de calibración se generará a partir de los datos actuales (tal y como estén antes de abrir el asistente). Por último, la botonera. La botonera está compuesta por dos botones, el primero cancela la creación y cierra la ventana, el segundo ejecuta la validación del nombre del fichero y si es satisfactoria, crea el fichero según el *radio button* seleccionado.

3.4.9 Ventana de preferencias

Cuando se pulsa el botón de preferencias (situado en la 6ª posición del *toolbar*), se muestra una ventana a modo de pop-up (ventana emergente), situada en el centro de la ventana padre (la pantalla principal) correspondiente al menú de preferencias de usuario. La pantalla de preferencias, abstrayéndonos a lo que es en sí la pantalla, tiene algunas curiosidades. Como he mencionado, la pantalla se muestra a modo de pop-up y está adjuntada a su ventana padre, por lo que siempre aparecerá superpuesta a la pantalla principal, además, tiene las propiedades de un *modal*, esto significa, que mientras se esté mostrando la pantalla, cualquier acción fuera de ésta queda invalidada, por lo que no se podría cerrar la aplicación sin guardar o cancelar los ajustes realizados en la pantalla.

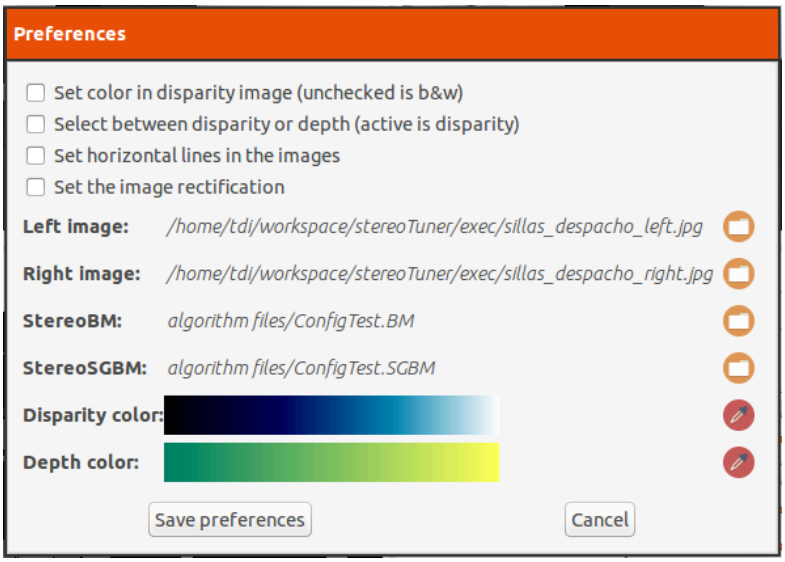


Figura 14. Ventana de preferencias.

En la ventana de preferencias se distinguen cuatro bloques estructurales: el primer bloque está constituido por cuatro *checkboxes*; el segundo, está formado por dos literales informando de la ruta hasta un archivo y dos botones; el tercero, lo forman dos imágenes y dos botones; y el último contiene la botonera de acciones, compuesta de dos botones. El primer bloque contiene los *checkboxes*, un *checkbox* es una clase hija de los botones de tipo *toggle*, esto significa que aunque no tengan el mismo aspecto, comparten funcionalidad, por lo que un *checkbox* es un botón con estado (puede estar activo o inactivo). Desde este primer bloque, se controlan las funciones respectivas al tercer grupo del *toolbar*. El segundo bloque muestra la ruta hasta cada imagen del par estéreo y el botón que les sigue (con el icono mnemónico de una carpeta), abre la ventana de selección de archivos (3.4.10). El tercer bloque, muestra los mapas de color seleccionados para pintar el mapa de disparidad, existiendo una diferente para los dos modos (disparidad y profundidad) y el botón que les sigue, abre la ventana de selección de mapas de color (3.4.11). El cuarto y último grupo, contiene 2 botones: el primer botón guarda las preferencias que se han modificado, éste es el único punto en el que se modifica el perfil de configuración (archivo .st), por lo que al actualizar las preferencias, la próxima vez que accedamos a la aplicación las propiedades estarán exactamente igual que como las hayamos guardado; el segundo botón cancela los cambios, es decir, deja el perfil de configuración en el mismo estado que tenía antes de pulsar el botón de preferencias.

3.4.10 Selector de archivos

La pantalla de selección de archivos se invoca desde varios puntos de la aplicación, nos sirve para seleccionar un perfil de configuración o para elegir el par de imágenes estéreo. Ésta pantalla es una pantalla de sistema ligeramente modificada.

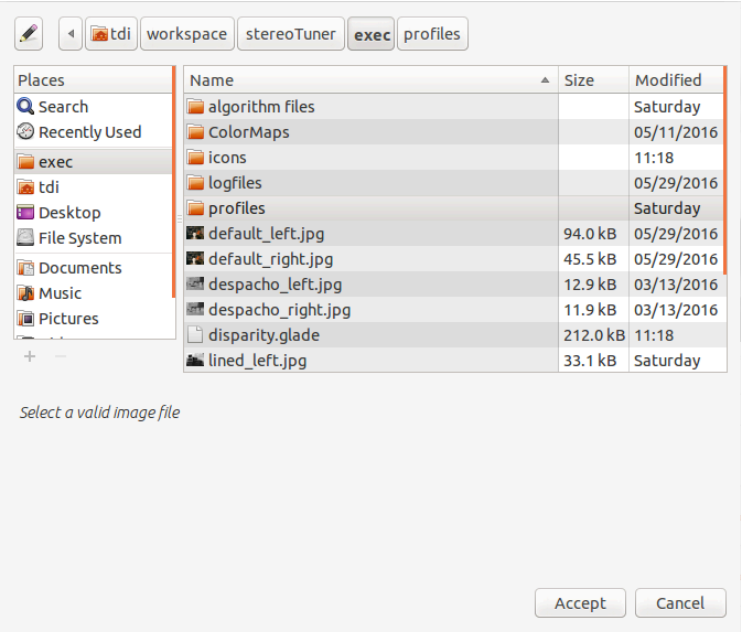


Figura 15. Ventana de selección de archivos.

Como se puede observar en la figura 15, esta ventana no tiene ni la estructura, ni el aspecto del resto de las ventanas de la aplicación, esto es debido a que no es propiamente una ventana de la aplicación, es un inspector de archivos de Ubuntu, ligeramente modificado. La ventana se compone de un árbol de archivos (situado en la parte superior izquierda), un inspector de archivos, un literal de aviso (inicialmente "Select a valid image file") y los botones de aceptar y cancelar la carga. Por leve modificación, me refiero al añadido del literal de aviso y a que no existe una entrada de texto para introducir el nombre de un archivo. El literal de aviso, es fundamental en esta ventana, ya que nos muestra el resultado de la validación del archivo seleccionado. Dependiendo desde el punto en el que estemos accediendo a esta ventana, querremos cargar un tipo de archivo u otro, si accedemos desde la ventana de preferencias, el selector sólo dejará cargar imágenes, si accedemos desde el asistente de perfiles (3.4.12), sólo dejará cargar perfiles de configuración (archivos st). En el caso de que la validación resulte errónea, se mostrará en rojo un mensaje de error.

En la carga de imágenes, la aplicación soporta muchos formatos, tantos como permita el objeto GtkImage. Para conocer la lista de formatos soportados, se invoca la función gdk_pixbuf_get_formats(), esta función devuelve un objeto lista (GSList *), cuyos elementos son estructuras del tipo GdkPixbuf. Se itera cada elemento extrayendo la propiedad *name*, la cual contiene el nombre del formato. A diferencia del error de validación que se muestra en la carga de los perfiles, en este caso no se imprimen todas las extensiones soportadas (la lista es extensa), se muestra un mensaje informando del fallo y se remite a la documentación. Como curiosidad, en la pantalla de ayuda rápida (3.4.13), uno de los *tips* se corresponde con la lista de extensiones soportadas.

3.4.11 Selector de mapas de color

La pantalla de selección de mapas de color, tiene una estructura muy sencilla, es un listado de *radio buttons* cuyo literal es el nombre y la imagen del mapa de color correspondiente. Inicialmente, aparece seleccionado el mapa de color propio a la propiedad desde la que hemos accedido, si hemos accedido desde "Disparity color", aparecerá el mapa de color asociado a la disparidad, si lo hacemos desde "Depth color", aparecerá el mapa asociado a la profundidad.

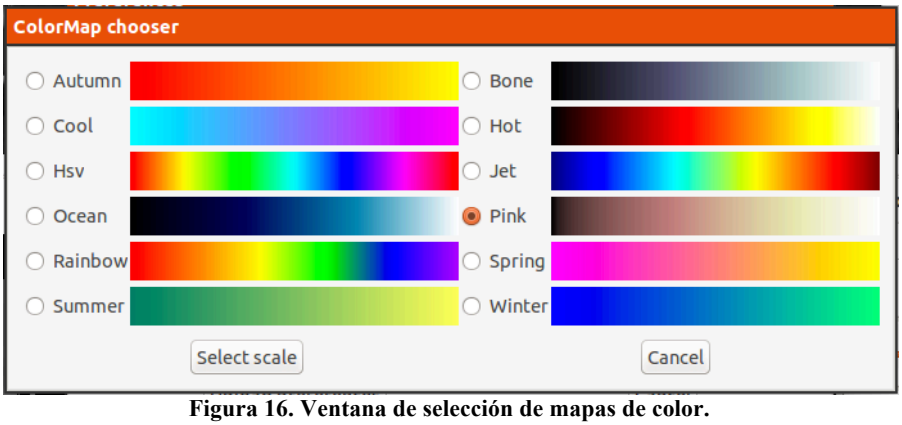


Figura 16. Ventana de selección de mapas de color.

Los *radio buttons* forman un único grupo, por lo que sólo puede estar seleccionado activamente un mapa. En la parte inferior de la lista está la botonera. El primer botón selecciona la escala y la carga en la ventana de preferencias y el segundo botón cierra la ventana y cancela cualquier cambio realizado.

3.4.12 Asistente de perfiles de configuración

La ventana del asistente de perfiles de configuración tiene la capacidad de cargar o crear un fichero de configuración, por lo tanto, una vez se realiza la acción, la aplicación recarga todos los datos, perdiéndose cualquier cambio que no se haya previamente guardado desde la ventana de preferencias.

El cuerpo de la ventana contiene: un literal con la ruta hacia el perfil actual (en el caso de existir) y un botón para acceder al selector de archivos; una entrada de texto para introducir el nombre del nuevo perfil de configuración y un botón para crearlo; un literal escondido justo debajo de la entrada, que se muestra en caso de que falle la validación que se lanza al crear el nuevo perfil. Hay que tener en cuenta que hay caracteres que no se pueden utilizar en nombres de archivos o carpetas (definidos por el SO), la validación impide llegar a este fallo y no permite el uso de estos caracteres, detallaré las validaciones en stereoTuner en el punto 3.6.13; y en la parte inferior, un botón de cierre.

Figura 17. Ventana del asistente de perfiles de configuración.

Como ya he comentado en la introducción a stereoTuner (3.3), y como detallaré en las funcionalidades de la aplicación (3.6.1), esta pantalla se muestra por defecto la primera vez que se arranca la aplicación o cuando el **xml** de configuración detecte nula la entrada del perfil. En este caso, el botón de cerrar "Close", se inhabilita, ya que el usuario no puede salir de esta ventana si no ha creado un perfil o cargado uno existente.

En el momento en el que se abre el seleccionador de archivos y se carga un perfil, el literal de ruta se modifica con la ruta hacia el perfil seleccionado y al lado del botón de cierre aparece un botón de carga "Load", que al clicarse hace efectiva la carga del fichero seleccionado.

3.4.13 Pantalla de ayuda rápida – Tips

La ventana de ayuda rápida contiene 4 únicos elementos: en la parte central y ocupando la mayor parte de la ventana, hay una imagen con el *tip* que se esté visualizando en ese momento,

en la parte inferior hay una botonera. La botonera está compuesta por dos botones que avanzan hacia el *tip* siguiente o hacia el anterior y un botón para cerrar la ventana.

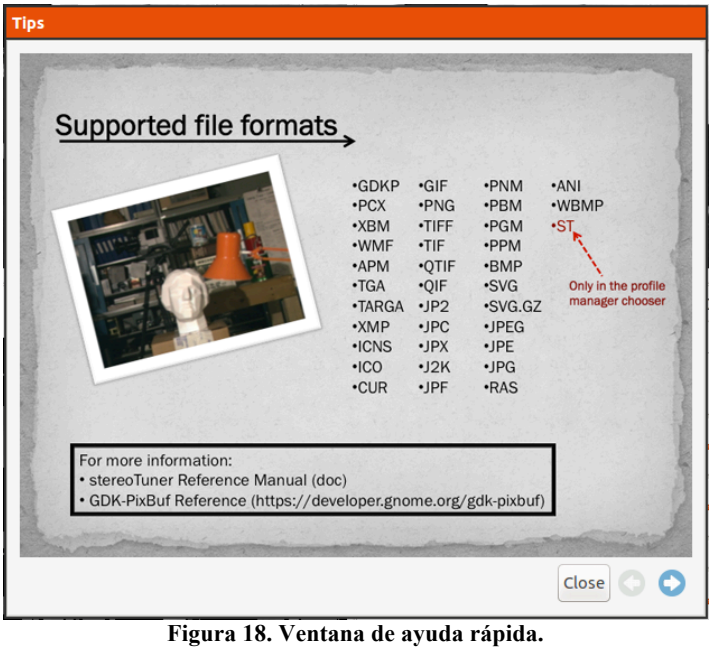


Figura 18. Ventana de ayuda rápida.

3.4.14 Pantalla de vídeo de ayuda

Estructuralmente, esta ventana no ofrece nada nuevo puesto que es idéntica a la de ayuda rápida (figura 18). La única diferencia, es que en lugar de una imagen central, contiene el archivo de vídeo a reproducir. La razón de no modificar la botonera, atiende a la posibilidad de que en un futuro se quisieran cargar varios videos, por lo que sigue conteniendo tres botones: cerrar ventana, vídeo anterior y vídeo siguiente. Pese a que estructuralmente, esta pantalla es muy sencilla, funcionalmente es muy compleja, debido a que GTK no tiene herramientas para reproducir vídeos, se tuvo que introducir otra librería. En el punto 3.6.11 desarrollaré en mayor profundidad esta funcionalidad.

3.4.15 Pantalla de exportación

La última ventana, es la ventana de exportación del *logger*. Esta ventana es la más sencilla de la aplicación, el cuerpo de la ventana contiene una entrada de texto, para que el usuario introduzca el nombre que le quiere dar al perfil de configuración y debajo, dos botones, uno para crear el fichero y otro para cancelar y cerrar la ventana.

Figura 19. Ventana de exportación del logger.

Al igual que el asistente de perfiles, debajo de la entrada hay un literal oculto, que en caso de que se introduzca algún carácter no imprimible, muestra el error con la lista de caracteres inválidos.

3.5 Tecnologías y configuración

3.5.1 Lenguajes y librerías seleccionados

En stereoTuner se han empleado varios lenguajes de programación, librerías y patrones de diseño para llevar a cabo las diferentes funcionalidades.

1. **C++**: La aplicación está desarrollada mediante el lenguaje de programación orientado a objetos C++.
2. **OpenCV**: Todas las funciones relativas al procesado de imágenes en la aplicación, se han realizado con lalibrería **OpenCV**.
3. **GTK+**: La interfaz de usuario, es decir, la "capa visual" de la aplicación, se he desarrollado utilizando la librería **GTK**.
4. **Xml**: Los ficheros de configuración de la aplicación están descritos mediante **xml**, tanto para la información de los objetos de la interfaz gráfica, cómo para la carga de los perfiles de configuración, para el *logger* y para otras muchas posibilidades implantables en un futuro.
5. **Glade**: Aunque técnicamente no es una librería, ni un lenguaje de programación (es un programa), **glade** se ha utilizado para generar el **xml** con el contenido de los objetos de la interfaz gráfica. Este **xml** se carga en el código fuente mediante el constructor de **GTK** (gtk_builder).
6. **CMake**: La herramienta utilizada para compilar el código fuente y buscar y conectar las diferentes librerías a la aplicación, ha sido **CMake**.
7. **FFmpeg**: Para trabajar con vídeos en la aplicación, ha sido necesario el uso de una librería específica que pueda trabajar con **GTK** sin causar problemas de integración, aunque hay varias, se ha optado por la librería **FFmpeg**.
8. ST: Esta es la extensión que se ha creado para el programa, es un fichero **xml**, con etiquetas propias, utilizadas bajo el prefijo st, (<sT:xxxx> siendo "xxxx" el nombre de la etiqueta). En el punto 3.5.3 detallaré en profundidad este tipo de ficheros.
9. **Txt**: Los ficheros de exportación del *logger*, son ficheros de texto.
10. **Singleton**: Para realizar las cachés de la aplicación, se ha utilizado un patrón de diseño conocido como **singleton**. Un **singleton**, es una clase que sólo puede tener una instancia, es decir, que sólo se puede instanciar una vez, y cada vez que se llama a dicha clase, se llama a la misma instancia. Dentro de la estructura del programa, es fácil reconocer los **singleton**, opté por diferenciarlos mediante el sufijo "Singleton", como por ejemplo, el **singleton** utilizado para las preferencias: *preferencesSingleton.hpp*. En el punto 3.5.4 entraré en detalle acerca de este patrón.

3.5.2 Árbol de archivos

Antes de entrar en materia con las funcionalidades y la parte más técnica de la aplicación, es necesario describir las clases que componen la aplicación, así como la estructura de directorios y ficheros, para que cuando se nombren de aquí en adelante, podamos ubicarnos correctamente.

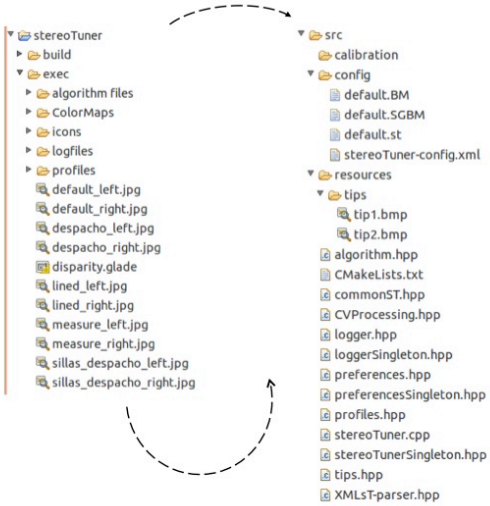


Figura 20. Árbol de archivos.

El directorio stereoTuner, es el directorio top del proyecto y es la carpeta que engloba a toda la aplicación. Colgando del top, distinguimos tres directorios: build, exec y src.

El directorio build, es la única carpeta que se puede eliminar sin causar problemas, es la carpeta que utiliza **CMake** para compilar el código fuente de la aplicación (en caso de no existir, tendríamos que crearla). Antes de ejecutar stereoTuner, es necesario compilar el código al menos una vez (si no seguimos desarrollando la aplicación), para ello se ejecuta la herramienta **CMake** apuntando al directorio fuente. En el directorio fuente, se encuentra el fichero CMakeLists.txt, el cual se utiliza para buscar las librerías necesarias (con sus versiones específicas) para ejecutar la aplicación (**OpenCV**, **GTK**, **FFmpeg**) y conectarlas a la aplicación, de modo que cuando llamemos a algún método de estas, nuestra máquina pueda encontrarlas.

El directorio exec contiene los ficheros necesarios para la ejecución del programa. Contiene las imágenes por defecto, el directorio con los iconos mnemónicos de la aplicación, el directorio con las imágenes de los mapas de color, el directorio donde se crean los ficheros de configuración (por lo tanto, el que los contiene), el directorio donde se crean los ficheros de calibración, el directorio dónde se exportan los ficheros del *logger* y contiene el archivo **glade** que define todos los objetos **GTK** de la aplicación.

El directorio src, contiene el código fuente de la aplicación. En su interior encontramos:

- El directorio *calibration*. Actualmente está vacío, formaría parte de una de las líneas futuras a desarrollar. En caso de que se desarrollase la posibilidad de procesar mapas de disparidad en tiempo real desde una cámara estéreo conectada al puerto USB, sería necesario calibrar dicha cámara antes de la rectificación de sus imágenes. Esta carpeta contendría los ficheros resultantes de dicha calibración.
- El directorio *config*. En su interior encontramos cuatro archivos:
    - default.BM. Este es el fichero de calibración por defecto para el algoritmo stereoBM.
    - default.SGBM. Este es el fichero de calibración por defecto para el algoritmo stereoSGBM.
    - default.st. Este es el perfil de configuración por defecto, contiene las rutas a las imágenes por defecto, todas las propiedades a *false*, los mapas de color por defecto (*ocean_scale* y *summer_scale*) y las rutas a los ficheros de calibración por defecto (default.BM y default.SGBM).
    - stereoTuner-config.xml. Este es el archivo de configuración de la aplicación, desde aquí se conecta el perfil de configuración.
- El directorio *resources*. En su interior se encuentra el directorio *tips*, este directorio alberga las imágenes **bmp**, debidamente escaladas (600x450), que se cargarán en la ventana *Tips* (3.4.13).
- El archivo algorithm.hpp. Contiene los métodos asociados a los ficheros de calibración, tanto a la lectura y escritura, como su creación. No obstante, no contiene los métodos que gestionan la información del estado del objeto **OpenCV** (BMState y SGBMState), ya que todos los métodos que manejan la librería **OpenCV** se encuentran en el archivo CVProcessing.hpp. El archivo contiene también la estructura CalibrationPanel, esta estructura contiene acceso a otras estructuras, como la estructura con los datos de **OpenCV** (CVdata), la estructura con las tablas de los algoritmos (Table_data), la estructura del selector de archivos (FileChooser), la estructura con las barras de ajuste del algoritmo stereoBM (BM_adjustment_bar_set), las del algoritmo estereoSGBM (SGBM_adjustment_bar_set) y contiene los objetos de la pantalla de creación de ficheros de calibración (3.4.8).
- El archivo CMakeLists.txt. Se usa para compilar el código. Situándose en la carpeta build, se ejecuta por consola el comando *cmake ../src*. **CMake** busca dentro de la carpeta seleccionada el archivo CmakeLists, en el se encuentran las librerías que tendrá que buscar **CMake** y que serán necesarias para ejecutar la aplicación, una vez que se resuelve con éxito el proceso, se ejecuta el comando *make*, que compila el código y conecta las librerías descritas en el archivo. Este archivo NO contiene las librerías, sino

que indica las librerías que son necesarias y si procede, dónde se encuentran localizadas.

- El archivo commonST.hpp. Contiene la definición de los colores y los métodos de uso general en la aplicación, como por ejemplo, el método que obtiene los formatos de imagen soportados en stereoTuner, *getSupportedFormats()*.
- El archivo CVProcessing.hpp. Contiene toda la lógica de **OpenCV**. El motivo de congregar todos los métodos que utilizan la librería **OpenCV** en un archivo único, es importante, ya que si se quisiera cambiar la librería de procesado de imágenes por otra, sólo se tendría que modificar los métodos de esta archivo. Este archivo contiene también las estructuras CVdata, Table_data, Table_data_and_data, BM_adjustment_bar_set y SGBM_adjustment_bar_set. CVdata contiene los objetos **OpenCV** y **GTK** que intervienen en el procesado del mapa de disparidad, como los objetos de las tres imágenes, los botones selectores de algoritmo, las imágenes Ipl…etc. Table_data, contiene las tablas de cada algoritmo. Table_data_and_data, contiene la estructura Table_data y la estructura CVdata. BM_adjustment_bar_set, contiene los objetos **GTK** de los ajustes del panel de calibración correspondiente a la tabla del algoritmo BM (*block-matching*) y SGBM_adjustment_bar_set, contiene los objetos **GTK** del panel de calibración correspondiente a la tabla del algoritmo SGBM (*semiglobal matching*).
- El archivo logger.hpp. Contiene los métodos respectivos a la pestaña del *notebook* perteneciente al *logger* (3.4.7) y a la ventana de exportación (3.4.15). También contiene la enumeración del número y tipos de columnas de la tabla del logger y las estructuras LoggerTreeView y ExportLogger. LoggerTreeView contiene los objetos **GTK** contenidos en la pestaña del *notebook* y ExportLogger los objetos **GTK** contenidos en la ventana de exportación.
- El archivo loggerSingleton.hpp. Este es uno de los tres **singleton** que se han definido en la aplicación. En su interior encontramos los mapas pertenecientes al cuerpo de los *logs* y al tiempo en el que se han emitido, así como los métodos necesarios para operar con ellos. En el punto 3.5.4 se detalla con precisión el patrón de diseño y la estructura que se ha elegido para definirlo en la aplicación. Como apunte, un mapa es un objeto de la clase std::string que guarda un registro asociado a dos valores, en nuestro caso un *int* y un *string*. La definición de una mapa es: std::map<Object key, Object value>. La diferencia con un buffer, radica en que el primer objeto *key* ejerce de identificador para el segundo objeto *value*, por tanto, no revela la posición del objeto *value* en el mapa, sino que identifica el objeto *value* con la que fue introducido. De modo que si se le pide al mapa el objeto *value* asociado con este identificador *key* (ya sea un valor entero, una cadena u otros objetos), devolverá el objeto *value* asociado (en el caso de que exista en el mapa, si no devolverá un valor nulo).
- El archivo preferences.hpp. Contiene todos los métodos referentes a la ventana de preferencias (3.4.9), a la carga y la validación de la configuración (al inicio y durante la ejecución), a la ventana de selección de archivos (3.4.10), a la ventana de selección de mapas de color (3.4.11) y las estructuras: FileChooser (asociada al selector de archivos), Colors (asociada al selector de mapas de color) y Preferences (asociada a la ventana de preferencias).
- El archivo preferencesSingleton.hpp. Es otro de los **singleton**, contiene todas las variables asociadas a las propiedades de la configuración: nombre del perfil de configuración, ruta a la imagen izquierda, ruta a la imagen derecha, si está activa la propiedad de las líneas de rectificación, si se debe mostrar la disparidad o la profundidad, si se debe mostrar el mapa de disparidad en escala de grises o mediante el mapa de color seleccionado, el código del mapa de color seleccionado y el mapa con los formatos de imagen soportados en la aplicación.
- El archivo profiles.hpp. Contiene los métodos referentes a la ventana del asistente de perfiles (3.4.12) y la estructura ProfileChooser, que contiene los objetos **GTK** propios de la ventana.

- El archivo stereoTuner.cpp. Este es el archivo principal, el que contiene el método *main()*. Desde aquí se incluyen todas las librerías (tantos las creadas para stereoTuner como las que no), contiene la estructura MainWidgets, que contiene la mayoría de los objetos comunes de la pantalla principal, contiene la estructura ToolBar, que contiene los botones de la barra de herramientas, contiene la super-estructura ChData, que contiene todas las estructuras de stereoTuner, contiene los métodos que actualizan la configuración o que modifican el aspecto de la pantalla principal, los métodos que se ejecutan al dispararse los eventos en cada *eventbox* y contiene el método main(). Desde el método main(), se inicializa toda la aplicación, se carga la configuración, se definen todas las estructuras y objetos, se crea el constructor de **GTK** a partir del archivo generado por **glade**, se definen las señales y los métodos a los que apuntan cuando las active el usuario, etc.
- El archivo stereoTunerSingleton.hpp. Este es el último de los **singleton**, generalmente contiene las propiedades usadas en la herramienta capture: si está activa, si se ha elegido valor en la imagen izquierda, el valor del pixel y la disparidad fijados en la imagen izquierda, el color de las líneas de medida, el mapa con los *tips* (las imágenes que se cargan en la ventana de ayuda rápida), etc.
- El archivo tips.hpp. Contiene todos los métodos y la estructura Tips, referentes a la ventana de ayuda rápida (3.4.13) y a la ventana de vídeo de ayuda (3.4.14).
- El archivo XMLsT-parser.hpp. Es el *parser* desarrollado para la lectura y escritura en el **XML** y en los perfiles de configuración, conteniendo el conjunto de métodos necesarios para ello. En el siguiente punto (3.5.3), entraré en detalle acerca de la configuración de la aplicación, por lo que explicaré algunas de las funciones más importantes del parser.

Otro aspecto importante del código de la aplicación, es la nomenclatura que se ha seguido para nombrar los métodos de stereoTuner. Se han seguido dos normas:

- Para los métodos genéricos de las librerías de stereoTuner: *commonST.hpp* y *XMLsT-parser.hpp*, se ha seguido la regla **CamelCase**, específicamente *lowerCamelCase*. **CamelCase** es un estilo de escritura que se utiliza para escribir frases, la idea es juntar la frase en una misma palabra, colocando la primera letra de cada palabra en mayúscula y el resto de letras en minúscula. En el caso del *lowerCamelCase*, se sigue esta regla a excepción de la primera palabra de la frase, que se coloca en minúscula. Así por tanto, si queremos escribir un método que devuelva todos los formatos, lo escribiríamos como: *devuelveTodosLosFormatos()*.
- Para el resto de métodos de la aplicación, se ha seguido una estructura propia para stereoTuner, es la siguiente: st_"nombre de la clase"_"método"(). Por ejemplo, el método que carga la configuración en stereoTuner, se encuentra en la librería preferences.hpp, su firma es: *st_preferences_read_config()*.

He realizado esta distinción debido a que commonST.hpp y XMLsT-parser.hpp se podrían utilizar desde cualquier punto de la aplicación, incluso desde otras aplicaciones ajenas a stereoTuner, ya que no dependen ni de **GTK**, ni de **OpenCV**, ni de **FFmpeg**.

3.5.3 Configuración de stereoTuner

Al arrancar stereoTuner, una de las primeras acciones que se lleva a cabo es la de cargar la configuración de la aplicación. Esto se realiza mediante el método *st_preferences_read_config()* ubicado en el fichero *preferences.hpp*:

```
static void st_preferences_read_config() {
       preferences& prefs = preferences::instance();

       //Get the profile name, if it's null all data are taken from the default file
       char *profile = getProfile();
       if (profile == NULL || profile == '\0' || strcmp(profile, " ") == 0 || strcmp(profile, "NULL") == 0 )
{
         //Nothing to do
       } else {
         prefs.set_profileName(profile);
       }
```

```
        //Get XML data with XMLsT parser
        prefs.set_leftFilename(getElementValueByAttribute("left_filename"));
        prefs.set_rightFilename(getElementValueByAttribute("right_filename"));

        if(strcmp(getElementValueByAttribute("lines"), "true") == 0) prefs.set_lines(true);
        else if(strcmp(getElementValueByAttribute("lines"), "false") == 0) prefs.set_lines(false);
        else prefs.set_lines(false);

        if(strcmp(getElementValueByAttribute("rectification"), "true") == 0)prefs.set_rectification(true);
        else if(strcmp(getElementValueByAttribute("rectification"), "false") == 0)
prefs.set_rectification(false);
        else prefs.set_rectification(false);

        if(strcmp(getElementValueByAttribute("disparity_depth"), "true") == 0) prefs.set_dispOrDeep(true);
        else if(strcmp(getElementValueByAttribute("disparity_depth"), "false") == 0)
prefs.set_dispOrDeep(false);
        else prefs.set_dispOrDeep(false);

        if(strcmp(getElementValueByAttribute("set_color"), "true") == 0) prefs.set_setColor(true);
        else if(strcmp(getElementValueByAttribute("set_color"), "false") == 0) prefs.set_setColor(false);
        else prefs.set_setColor(false);

        prefs.set_disparityColor(fillColorByName(getElementValueByAttribute("disparity_color")));
        prefs.set_deepColor(fillColorByName(getElementValueByAttribute("deep_color")));

        prefs.set_stereoBM(getElementValueByAttribute("stereoBM"));
        prefs.set_stereoSGBM(getElementValueByAttribute("stereoSGBM"));

        //Set all supported formats into preferences formats map
        getSupportedFormats();

}
```

Código 1. Carga de la configuración.

Siguiendo el código descrito, primero se carga el perfil de configuración mediante el método *getProfile()* ubicado en el *parser* desarrollado para leer y escribir en los ficheros de configuración y calibración de la aplicación: *XMLsT-parser.hpp* (**xml**, **st**, **BM** y **SGBM**). Desde aquí, se accede al fichero de configuración *stereoTuner-config.xml* y se extrae la ruta hacia el perfil de configuración (fichero **st**). En este punto, pueden ocurrir dos cosas, o bien que se encuentre una ruta hacia un archivo válido, por lo que se carguen todas las propiedades de forma correcta, o que no se encuentre un fichero válido, con lo que todas las propiedades se rellenen con los valores por defecto extraídos del fichero default.st, a excepción de la propiedad profile, que continuará nula. Por último, se llama al método *getSupportedFormats()* de *commonST.hpp* que extrae la lista de todas las extensiones de archivos válidas para la carga de imágenes en la aplicación.

Para situarnos mejor, voy a describir la estructura del fichero general de configuración *stereoTuner-config.xml*, de los perfiles de configuración (archivos **st**), de los ficheros de calibración (archivos **BM** y **SGBM**) y del *parser XMLsT-parser.hpp*.

El fichero de configuración desde el cual se obtiene la información básica, está implementado al mínimo de su potencial. Con esto me refiero, a que las posibilidades que ofrece son abrumadoras, actualmente se utiliza para "enganchar" el perfil de configuración, aunque se podría utilizar para muchísimas cosas más. Algunas de ellas:

- Secuenciar ficheros exportados del *logger* y de ese modo, arrancar la aplicación con los *logs* de la última sesión en el mismo punto en el que se quedaron.
- Enganchar la ruta a los ficheros de calibración correspondientes a la cámara estéreo introducida en el puerto USB (en el caso de que se desarrollara la funcionalidad del cálculo del mapa de disparidad en tiempo real).
- Enganchar la ruta a los ficheros de rectificación (con las matrices fundamental y esencial), para rectificar las imágenes del perfil de configuración antes de empezar la aplicación.

El fichero de configuración se compone de dos etiquetas principales: <sT:configuration> y <sT:config-element>. <sT:configuration> no admite atributos y sólo contiene dos propiedades: el nombre (<filename>) y la versión (<version>) del archivo. <sT:config-element> en cambio, sí que admite un atributo, el identificador del elemento en cuestión (*id*), y admite la propiedad <name>, la cual "engancha" la ruta al perfil de configuración.

```
<?xml version="1.0" encoding="windows-1250"?>
<sT:configuration>
  <filename>stereoTuner-config.xml</filename>
  <version>1.0</version>

  <sT:config-element id="profile">
    <name>/home/tdi/workspace/stereoTuner/exec/profiles/stereoTest.st</name>
  </sT:config-element>

  <sT:config-element id="log">
    <name>NULL</name>
  </sT:config-element>

</sT:configuration>
```

Código 2. Fichero de configuración.

Para extraer los valores de las etiquetas, se utilizan los métodos del fichero *XMLsT-parser.hpp*. Por ejemplo, para obtener el <sT:config-element> con el atributo profile, se utiliza el método *getProfile():*

```
static char *getProfile() {
	preferences& prefs = preferences::instance();
	string line;
	char *result;
	bool search_val = false;
	ifstream in("../src/config/stereoTuner-config.xml");
	while (getline(in, line)) {
	  string tmp, tmp_name, tmp_val;
	  if (!search_val) {
		tmp_name = getTagAttribute(line, "<sT:config-element", "id=");
	  } else {
		if(getEndTag(line, "</sT:config-element>")) {
			return NULL;
		}
		tmp_val = getTag(line, "<name>");
	  }
	  //Now we have the required element, it's time to search the value
	  if (tmp_name.length() != 0 && tmp_name == "profile") {
		search_val = true;
	  }
	  //We catch the value and return the value
	  if (tmp_val.length() != 0) {
		result = strToChar(tmp_val);
		search_val = false;
	  }
	}
	return result;

}
```

Código 3. Obtención del perfil de configuración.

Este método carga el fichero de configuración en un *stream* y lo lee línea a línea, en cada línea busca el <st:config-element> con el atributo *id* de valor “profile”, una vez lo encuentra, busca la etiqueta <name> y devuelve su contenido.

Para obtener el valor del atributo *id*, le pasamos la línea que se está escaneando al método *getTagAttribute()*:

```
static string getTagAttribute(string line, string tag, string attribute) {
	bool begin_tag = false;
	bool begin_attribute = false;
	string tmp, tmp_val, tmp2, result;
	for (int i = 0; i < line.length(); i++) {
	  if (line[i] == ' ' && tmp.size() == 0) {
		//Nothing to do
	  } else {
		tmp += line[i];
	  }
	  if (tmp == tag) {
		begin_tag = true;
		i++; //Jump to next cycle to avoid the > character
	  }
	  if (begin_tag && line[i] == '>') {
		begin_tag = false;
	  }
	  if (begin_tag) {
		tmp_val += line[i];
	  }
	}
	for (int i = 0; i < tmp_val.length(); i++) {
	  if (tmp_val[i] == ' ' && tmp2.size() == 0) {
		//Nothing to do
	  } else {
```

```
                tmp2 += tmp_val[i];
        }
        if(tmp2 == attribute) {
                begin_attribute = true;
                i+=2; ////Jump two cycles to avoid the =" characters
        }
        if (begin_attribute && tmp_val[i] == '\"') {
                begin_attribute = false;
        }
        if (begin_attribute) {
                result += tmp_val[i];
        }
      }

      return result;
}
```

Código 4. Obtención del valor de un atributo.

Una vez se ha encontrado el objeto <st:config-element> con el atributo especificado, se busca obtener el valor de la etiqueta <name>, para ello, le pasamos la línea que estamos escaneando al método *getTag()*:

```
static string getTag(string line, string tag) {
      bool begin_tag = false;
      string tmp, tmp_val;
      for (int i = 0; i < line.length(); i++) {
       if (line[i] == ' ' && tmp.size() == 0) {
           //Nothing to do
       } else {
           tmp += line[i];
       }
       if (tmp == tag) {
           begin_tag = true;
           i++; //Jump to next cicle to avoid the > character
       }
       if (begin_tag && line[i] == '<' && line[i + 1] == '/') {
           begin_tag = false;
       }
       if (begin_tag) {
           tmp_val += line[i];
       }
   }
      return tmp_val;

}
```

Código 5. Obtención del valor de una etiqueta.

El método *getTag()* va almacenando carácter a carácter (siempre que no sea nulo, o un espacio) en un buffer (tmp) y lo va comparando con la etiqueta que se busca, en nuestro caso <name>, una vez la encuentra, guarda en otro buffer toda la información que haya hasta el cierre de la etiqueta (</name>) y devuelve el resultado del segundo buffer (tmp_val). El caso de *getTagAttribute()*, es una ampliación de *getTag()*, ya que una vez se ha encontrado el *tag*, se busca el atributo y se devuelve el contenido de las comillas (atributo = ”valor”). Para asegurarnos de que no nos pasamos de línea y empezamos a leer las etiquetas de otro elemento, se usa el método *getEndTag()*, que nos mantiene dentro del elemento seleccionado y que en caso de llegar al cierre del elemento (se entiende por tanto que no se ha obtenido la etiqueta solicitada) se retorna un valor nulo.

```
static bool getEndTag(string line, string tag) {
      bool begin_tag = false;
      string tmp, tmp_val;
      for (int i = 0; i < line.length(); i++) {
        if (line[i] == ' ' && tmp.size() == 0) {
        //Nothing to do
        } else {
                tmp += line[i];
        }
        if (tmp == tag) {
                begin_tag = true;
        }
      }
      return begin_tag;

}
```

Código 6. Obtención del cierre de una etiqueta.

Cuando desde el asistente de perfiles se carga un perfil, se sobrescribe el fichero de configuración para poner la ruta al perfil seleccionado, para ello se invoca al método del *parser setNameByAttribute()*.

```
static void setNameByAttribute(string name, string value) {
	preferences& prefs = preferences::instance();
	string line;
	string spacing = "    "; //value tag has a spacing of 4 characters
	bool search_val = false;
	bool begin_tag = false;
	bool copy = true;
	ifstream in("../src/config/stereoTuner-config.xml");
	ofstream out("../src/config/file-out.xml");
	while (getline(in, line)) {
	  string tmp, tmp_name, tmp_val;
	  if (!search_val) {
		tmp_name = getTagAttribute(line, "<sT:config-element", "id=");
	  } else {
		tmp_val = getTag(line, "<name>");
	  }
	  //Now we have the required element, it's time to search the value
	  if (tmp_name.length() != 0 && tmp_name == name) {
		search_val = true;
	  }
	  //We catch the value and set into preferences singleton class
	  if (tmp_val.length() != 0 && copy) {
		out << spacing << "<name>" << value << "</name>" << endl;
		copy = false;
	  } else {
		out << line << endl;
	  }
	}
	//Rename the output file to overwrite the input file and complete the upload process
	rename("../src/config/file-out.xml", "../src/config/stereoTuner-config.xml");

}
```

Código 7. Escritura de la etiqueta *name* a partir del valor de un atributo.

Para escribir la ruta entre las etiquetas <name> y </name>, se abre un fichero temporal llamado file-out.xml, donde se irá copiando el contenido del fichero de configuración, al mismo tiempo que se busca el <sT:config-element> con el *id* proporcionado. Cuando se encuentra, se busca la etiqueta <name> y se reemplaza la línea entera con la etiqueta, la ruta al perfil de configuración y el cierre de la etiqueta. Por último se renombra el fichero temporal con el mismo nombre que el fichero de configuración para sobrescribirlo.

Desde el fichero de configuración, se pasa al perfil de configuración, que es donde se encuentra toda la información necesaria para ejecutar la aplicación. El perfil de configuración tiene dos tipos de etiquetas: <sT:preferences> y <sT:element>. <sT:preferences> es la etiqueta top que engloba a las demás, esta etiqueta, admite el atributo *name* y tiene dos propiedades, el nombre (<filename>) y la versión (<versión>) del archivo, además, admite las etiquetas <sT:element>. Las etiquetas <sT:element> admiten un atributo y dos propiedades: *name*, <type> y <value>. El atributo *name* contiene el nombre del elemento, la etiqueta <type> indica el tipo de elemento y la etiqueta <value> contiene el valor del elemento. Por ejemplo, para el elemento que contiene la imagen izquierda, el valor del atributo será *left_filename*, el valor de la etiqueta <type> será *file* y el valor de la etiqueta <value> será la ruta hasta la imagen.

```
<?xml version="1.0" encoding="windows-1250"?>
<sT:preferences>
  <filename>default.st</filename>
  <version>1.0</version>

  <!-- Elements -->
  <sT:element name="left_filename">
    <type>file</type>
    <value>default_left.jpg</value>
  </sT:element>

  <sT:element name="right_filename">
    <type>file</type>
    <value>default_right.jpg</value>
  </sT:element>

  <sT:element name="lines">
    <type>checkbox</type>
    <value>false</value>
  </sT:element>
```

```
  <sT:element ="rectification">
    <type>checkbox</type>
    <value>false</value>
  </sT:element>

  <sT:element name="disparity_depth">
    <type>checkbox</type>
    <value>false</value>
  </sT:element>

  <sT:element name="set_color">
    <type>checkbox</type>
    <value>false</value>
  </sT:element>

  <sT:element name="disparity_color">
    <type>attribute</type>
    <value>ocean_scale</value>
  </sT:element>

  <sT:element name="deep_color">
    <type>attribute</type>
    <value>summer_scale</value>
  </sT:element>

  <sT:element name="stereoBM">
    <type>file</type>
    <value>../src/config/default.BM</value>
  </sT:element>

  <sT:element name="stereoSGBM">
    <type>file</type>
    <value>../src/config/default.SGBM</value>
  </sT:element>
</sT:preferences>
```

Código 8. Fichero del perfil de configuración.

Para obtener el valor de las etiqueta <value> de un <st:element>, se invoca el método *getElementValueByAttribute()*:

```
static char *getElementValueByAttribute(string attribute) {
       preferences& prefs = preferences::instance();
       string line;
       char *result;
       bool search_val = false;
       ifstream in;
       if (prefs.get_profileName() == NULL || prefs.get_profileName()[0] == '\0') {
          in.open("../src/config/default.st");
       } else {
          in.open(prefs.get_profileName());
       }
       while (getline(in, line)) {
          string tmp, tmp_name, tmp_val;
          if (!search_val) {
                   tmp_name = getTagAttribute(line, "<sT:element", "name=");
          } else {
                   if(getEndTag(line, "</sT:element>")) {
                            return NULL;
                   }
                   tmp_val = getTag(line, "<value>");
          }
          //Now we have the required element, it's time to search the value
          if (tmp_name.length() != 0 && tmp_name == attribute) {
                   search_val = true;
          }
          //We catch the value and set into preferences singleton class
          if (tmp_val.length() != 0) {
                   result = strToChar(tmp_val);
                   search_val = false;
          }
       }
       return result;

}
```

Código 9. Valor de un elemento a partir de su atributo.

Este método procede de manera parecida al método *getProfiles()*, sólo que en vez de buscar la etiqueta <name>, busca la etiqueta <value>.

Las últimas dos propiedades del perfil de configuración (código 8), son las que enganchan a los ficheros de calibración. La propiedad stereoBM, engancha al fichero de calibración correspondiente al algoritmo stereoBM, cuya extensión es **BM** y la propiedad stereoSGBM,

engancha al fichero de calibración del algoritmo stereoSGBM, cuya extensión es **SGBM**. Estos ficheros, definen el valor de cada barra de ajuste del panel de calibración y hay uno para cada algoritmo, aunque la estructura del fichero es la misma para ambos. En estos ficheros hay dos tipos de etiquetas: <sT:algorithm> y <sT:adjustment>. La etiqueta <sT:algorithm> es la etiqueta global que agrupa el resto de etiquetas del archivo, admite el atributo *type* (que describe el tipo de algoritmo, BM o SGBM) y las propiedades <filename> y <version>. La etiqueta <sT:adjustment> es idéntica a la etiqueta <sT:element>. El atributo *name* refleja el nombre de la barra de ajuste y las propiedades <type> y <value> reflejan el tipo (entero, decimal, booleano) y valor del dato.

Siguiendo el código 1, nos queda por describir que se hace cuando se captura el valor de un elemento, bien sea el <sT:config-element> con la ruta hasta el perfil, cualquiera de los <sT:element> con el valor de las distintas propiedades de las preferencias o cualquiera de los <sT:adjustment> con los valores de las barras de ajuste del panel de calibración. Para cargar todas estas propiedades obtenidas de los ficheros, inicializamos la instancia del objeto *preferences* y guardamos cada propiedad en su variable correspondiente. Como no podemos acceder a las variables de la clase directamente, porque son privadas, accedemos a los *getters* y *setters* que son públicos y sí pueden acceder a las variables privadas. En el siguiente punto (3.5.4) explicaré que es un **singleton**, su estructura y las ventajas de su uso, sobre todo, en la implementación de cachés en las aplicaciones.

3.5.4 Patrón de diseño singleton

El patrón **singleton**, es un patrón de diseño utilizado para impedir que una clase se instancie más de una vez. La palabra anglófona *singleton*, viene a significar "instancia única". La idea básica de este patrón de diseño, es asegurar que la clase implementada sólo pueda tener una instancia, dotando a los objetos que contiene, de un único punto de acceso. Para implementar el patrón **singleton** en el código de la aplicación, he seguido el patrón definido por Scott Meyers en 7.3 [10] (*The Meyers singleton*). Este diseño se basa en el uso de modificadores de acceso en la clase, de modo que una parte de la clase es de acceso privado (por lo que sólo se podrá acceder desde dentro de la clase) y la otra parte de la clase es de acceso público, a la cual podremos acceder desde cualquier parte del código siempre que hayamos declarado la instancia en el método que realice la llamada. Digo declarar la instancia, porque no sería del todo correcto utilizar la expresión "instanciar la clase", ya que esta clase, sólo se instancia una vez y cada vez que se genera un objeto de esta clase, es siempre de la misma instancia, con el siguiente ejemplo lo veremos más claro.

```
class Singleton {
    private:
        Object object;

        singleton() {
        }

    public:
        ~singleton() {
         //Do something
        }

        Object getObject() {
            return object;
        }

        void setObject(Object object) {
         this->object = object
        }

        static singleton& instance() {
            static singleton s;
            return s;
        }
};
```

Código 10. Patrón de diseño singleton.

Como se observa en el código 10, los objetos de la clase *Singleton*, se encuentran dentro del acceso privado, por lo que desde fuera, no podemos acceder a su valor. Para acceder a ellos, se implementan métodos que obtienen y/o modifican los valores de estos objetos (getters y setters). Para poder acceder a los métodos públicos de la clase, crearemos un objeto de la instancia y a partir de él, podremos modificar u obtener los objetos privados. En el siguiente código, se modifica el objeto privado de la clase, por uno nuevo:

```
singleton& mySing = singleton::instance();
Object newObject;

mySing.setObject(newObject);
```

Código 11. Acceso método público.

Uno de los usos más extendidos de este tipo de patrón es en la implementación de cachés. Es común en desarrollo de back-end encontrarse con este tipo de clases. En estos casos, suelen contener objetos de tipo mapa (std::map<identifier, Object>). Este tipo de objetos, se asemejan a un buffer, pero tienen una peculiaridad, van acompañados de un identificador único. Para que nos entendamos, imaginemos que tenemos una clase denominada *Triangulo*, cuyo contenido es el siguiente:

```
class Triangulo {
  public:
    int ladoA;
    int ladoB;
    int ladoC;
    void llenar(int a, int b, int c);
};

//Función que llena el triángulo
void Triangulo::llenar(int a, int b, int c) {
  ladoA = a;
  ladoB = b;
  ladoC = c;

}
```

Código 12. Clase triángulo.

Imaginemos que quisiéramos tener almacenado un triángulo de cada tipo. Si lo hiciéramos en un buffer, para obtener el triángulo equilátero, tendríamos que sacar los tres triángulos y comparar sus lados para saber cuál es el equilátero. Sin embargo, si empleamos un mapa, podemos identificar cada triángulo con su tipo y obtenerlo mediante su identificador, veamos un ejemplo:

```
int main () {
    Triangulo tri1, tri2, tri3;

    tri1.llenar(12, 12, 12);
    tri2.llenar(7, 8, 8);
    tri3.llenar (9, 10, 11);

    //Introducimos los triángulos en el mapa
    map<string, Triangulo> mapa;
    mapa["equilatero"] = tri1;
    mapa["isosceles"] = tri2;
    mapa["escaleno"] = tri3;

    //Leemos el triángulo equilátero
    Triangulo tr = mapa["equilatero"];
    cout << "El triángulo equilatero es: (" << tr.ladoA
    << "," << tr.ladoB << "," << tr.ladoC << ")." << endl;


}
```

Código 13. Extraer objeto de un buffer y de un mapa.

La salida del programa será: El triángulo equilátero es: (12,12,12).

He utilizado este patrón de diseño para diseñar las clases: preferences (preferencesSingleton.hpp), logger (loggerSingleton.hpp) y stereoTuner (stereoTunerSingleton.hpp). La razón de hacerlo de esta manera y no como variables globales, ha sido para proteger los datos de los objetos, ya que en las tres clases se maneja la información

más sensible de la aplicación y un mal uso de estas variables, podría provocar un fallo en la ejecución del programa, además por supuesto, de utilizar mapas para implementar la caché de los *logs*, de los formatos de las imágenes y de las rutas hacia las imágenes con los *tips*.

3.6 Funcionalidades

3.6.1 Empezando en stereoTuner ("Getting started")

Cuando arrancamos por primera vez la aplicación (no cada vez, si no una primera vez histórica), el **xml** de configuración aparece completamente vacío, eso quiere decir que el elemento *profile* contiene un valor nulo. Por lo tanto, cuando se ejecute la aplicación, el primer paso de la lectura de configuración fallará (al no encontrarse un perfil válido en el fichero de configuración) y el programa configurará todas las propiedades por defecto. Cómo los ficheros por defecto no son modificables por el usuario, una vez se ha configurado todo, se abrirá automáticamente el asistente de perfiles de configuración. En este punto, no podremos continuar hasta que no hallamos cargado un perfil de configuración existente (válido) o hayamos creado uno nuevo.

En el siguiente punto aumentaré la información acerca de la carga y creación de perfiles de configuración, así como de sus acciones principales y sus consecuencias sobre el código o los ficheros involucrados.

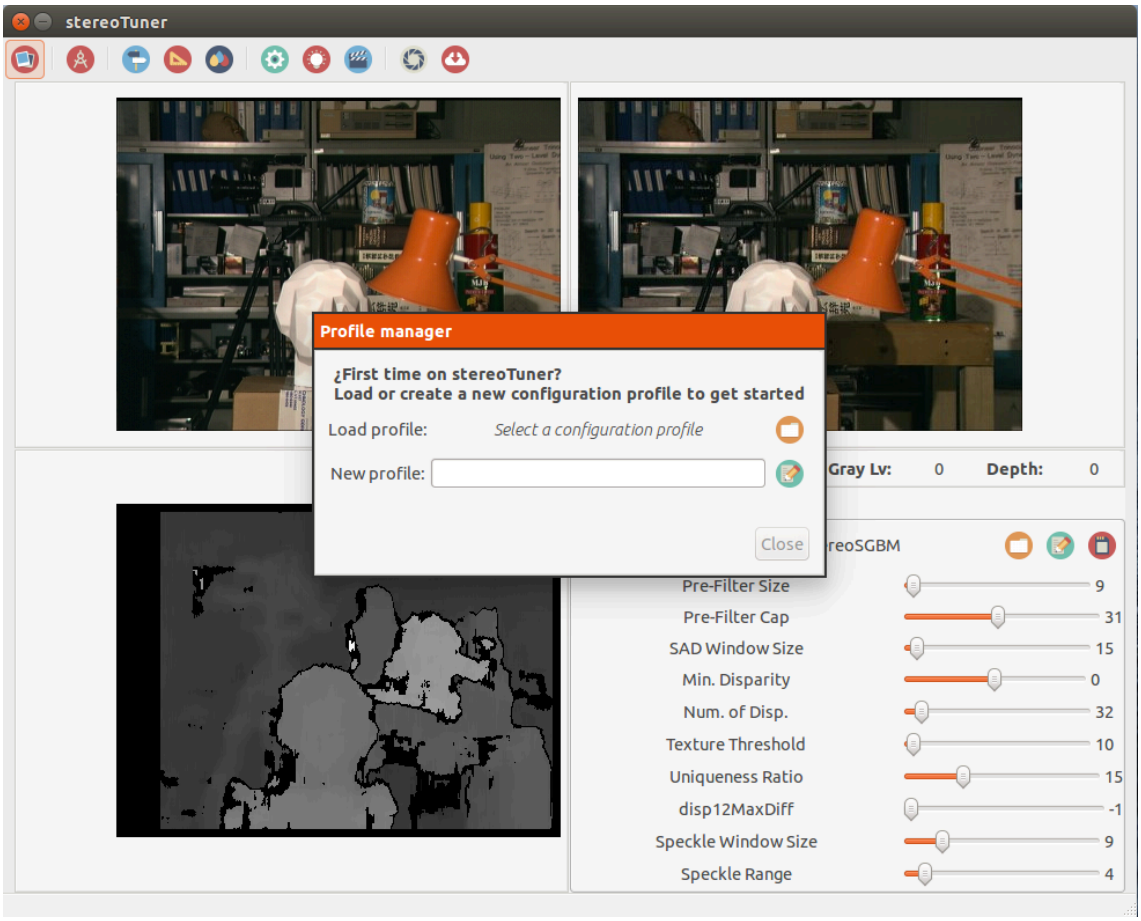


Figura 21. Inicio stereoTuner.

3.6.2 Creación y carga de perfiles de configuración

Cuando desde el asistente de perfiles de configuración (figura 17, punto 3.4.12) se crea o se carga un nuevo perfil, se llevan a cabo una serie de acciones que afectan a la ejecución de la aplicación, veamos cuáles son.

Para cargar un perfil de configuración, el usuario clica sobre el icono de la carpeta, acto seguido se abre la ventana de selección de archivos. Desde esta ventana, el usuario selecciona un fichero

y clica sobre el botón de aceptar, si el fichero seleccionado no tienen un extensión **st**, la ventana muestra un error y no realiza ninguna acción posterior hasta que se haya cargado un archivo válido o se pulse sobre el botón de cancelar, en este último caso se cerrará la ventana de selección. Si la validación es positiva, se cierra la ventana de selección y aparece la ruta al archivo seleccionado como valor de "Load profile:", también aparece un segundo botón junto al botón de cerrar la ventana, para ejecutar la carga del archivo.

Cuando el usuario clica el botón de carga de perfil desde el asistente de perfiles, se procede a un "reinicio total" de la aplicación, como no se puede re-ejecutar el método main(), se ejecutan todas las funciones que modifican los parámetros, a partir de las propiedades controladas por la clase *preferences* (preferencesSingleton). Se leen todas las propiedades del nuevo perfil y se escriben en las variables de la clase *preferences*, se vuelve a ejecutar la carga de los objetos **OpenCV**, se modifican todos los objetos necesarios en la ventana principal (cambio de las imágenes estéreo, estado de los botones de la barra de herramientas, valores en el panel de calibración…), así como la ejecución de todas las acciones que lleven asociadas y se ejecutan todos los procedimientos para el cálculo de la disparidad. Como he dicho, un "reinicio total".

Para crear un perfil de configuración, se introduce un nombre válido en la entrada de texto del asistente y se clica sobre el icono del lápiz. Si la validación es positiva, es decir, que no se hayan introducidos caracteres prohibidos (3.6.13), se ejecutan todas las acciones propias de la creación del perfil, si es negativa, se muestra un literal de error y no se realiza ninguna acción posterior. Al clicar el botón de creación (validación positiva) se ejecuta el método situado en el *parser*, que crea una copia idéntica del fichero por defecto (default.st), sólo modificando su nombre. Este fichero se sitúa dentro del directorio *profiles*, que a su vez está dentro del directorio de ejecución (*exec*). Una vez se ha creado el fichero, se ejecuta el "reinicio total" pero partiendo desde este fichero recién creado.

La modificación del perfil de configuración se lleva a cabo desde otros tres puntos: la actualización de las preferencias (3.6.9) desde la ventana de preferencias, la carga y creación de los ficheros de calibración. En el caso de la carga desde el menú del panel de calibración y en el caso de la creación, desde la ventana de creación de ficheros de calibración.

3.6.3 Cálculo del mapa de disparidad

stereoTuner tiene como objetivo principal el cálculo del mapa de disparidad, por lo que está funcionalidad es de máxima importancia en la aplicación. Para calcular la correspondencia entre las dos imágenes estéreo suministradas mediante las propiedades *left_filename* y *right_filename* del perfil de configuración, se emplean dos tipos de algoritmos: StereoBM y StereoGSBM.

El algoritmo StereoBM, calcula la correspondencia mediante el algoritmo por correspondencia entre bloques (*block-matching*) y se procesa mediante el método *st_cvprocessing_compute_stereoBM()* en el código de la aplicación. Este método le pasa a la función de **OpenCV** que calcula la correspondencia cvFindStereoCorrespondenceBM(), el buffer con el contenido de la imagen izquierda (cv_image_left), el de la imagen derecha (cv_image_right), el de la imagen de disparidad (cv_image_depth) y el objeto BMState (se crea al iniciar la aplicación en función de los valores del fichero de calibración). Este método de **OpenCV** carga en el buffer de la imagen de disparidad el contenido del nuevo mapa. A continuación, se normaliza el buffer con valores de 0 a 256 para poder visualizarlo en el *display*. Una vez normalizado, se convierte el buffer en una imagen **OpenCV** y una vez formada, se transforma la imagen **OpenCV** (IplImage) en una imagen **GTK** (GtkImage), para poder visualizarla en el objeto GtkImage *image_depth de la pantalla principal.

El algoritmo StereoSGBM, calcula la correspondencia mediante el algoritmo por correspondencia semi-global (*Semiglobal mathing*) y se calcula mediante el método *st_cvprocessing_compute_stereoSGBM()*. Para calcular la correspondencia se le pasa las variables Mat con el contenido de las tres imágenes (izquierda, derecha y disparidad) al SGBMState que se crea al inicio con los valores del fichero de calibración. SGBMState carga como salida el calculo del mapa de disparidad en la variable Mat asociada

(image_depth_SGBM). Una vez tenemos la variable Mat con el mapa de disparidad, la convertimos en un buffer de **OpenCV** (CvMat) y a partir de aquí se procede de igual manera que en el algoritmo StereoBM: se normaliza entre 0 y 256, se convierte en una IplImage, se transforma en una GtkImage mediante un GdkPixBuf y se sitúa en el objeto GtkImage de la pantalla principal, para que se pueda ver el resultado del cálculo.

Los métodos *st_cvprocessing_compute_stereoBM()* y *st_cvprocessing_compute_stereoSGBM()* es decir, el cálculo de la disparidad, se realizan desde varios puntos de la aplicación y como finalización de muchas acciones, veamos cuáles son:

- Al actualizar las preferencias. Independientemente de si se han modificado las imágenes, se ejecuta el cálculo de la correspondencia (sólo del algoritmo seleccionado).
- Cada vez que se cambia de algoritmo. Cada vez que se selecciona (mediante los *radio buttons*) un algoritmo distinto al que se está visualizando, se recalcula el mapa de disparidad.
- Cuando se carga o se crea un nuevo perfil de configuración. En el "reinicio total" de la aplicación, también se lleva a cabo el cálculo del mapa de disparidad.
- Al crear o cargar un nuevo fichero de calibración.
- Cuando se modifica cualquier ajuste en el panel de calibración. Al modificar el valor de una barra de ajuste del panel de calibración, se procesa de manera dinámica y se calcula de nuevo el mapa de disparidad con el nuevo valor de la propiedad que se haya modificado.

3.6.4 Ficheros de calibración

En stereoTuner hay varios tipos de archivos de configuración y cada archivo realiza una tarea específica. El fichero de configuración general stereoTuner-config.xml, sirve para dar acceso al perfil de configuración (archivo st) que será quién informe de todas las propiedades de la aplicación. Dos de las propiedades del perfil de configuración, enganchan a otro tipo de archivo, el fichero de calibración. Cada perfil de configuración puede estar sincronizado con un fichero de calibración por algoritmo, como hay implementados dos algoritmos, sólo se podrán sincronizar dos ficheros de calibración.

Los ficheros de calibración tienen la información de los valores de las barras de ajuste del panel de calibración asociadas a cada algoritmo. Para crear un fichero de calibración, el usuario clica sobre el botón con el icono de un lápiz, situado en el menú de calibración (notebook – "Algorithm"). Al clicar sobre el icono, se abre la ventana de creación de ficheros de calibración (3.4.8), desde la cual se introduce el nombre del fichero en la entrada de texto y se selecciona una de las dos opciones disponibles. Se puede elegir entre crear el fichero de calibración a partir de los datos por defecto (copiará el contenido de default.BM o default.SGBM), o crearlo a partir de los datos actuales del menú. Al clicar sobre el botón "Create", se crea el perfil de configuración, se carga en las propiedades del **singleton** de preferencias y se incluye en la propiedad correspondiente del perfil de configuración (se sobrescribe el archivo). Hay que tener en cuenta que la acción de crear, crea el fichero de calibración del algoritmo que está seleccionado en ese momento. Por lo tanto, si entramos en la ventana de creación desde el algoritmo StereoBM, el fichero de calibración resultante tendrá una extensión BM, si lo hacemos desde el algoritmo StereoSGBM, el fichero de calibración tendrá la extensión SGBM.

Para cargar un fichero de calibración, el usuario clica sobre el botón con el icono de una carpeta en el menú de calibración. Acto seguido, se abre la ventana de selección de archivos. Para cargar un fichero de calibración, es indiferente del tipo de algoritmo desde el que accedamos, ya que cuando se cargue un algoritmo, la aplicación mostrará los datos del algoritmo recién cargado en el menú de calibración. La única restricción que existe en esta acción, es la de la carga de un archivo válido, sólo se permite cargar archivos BM o SGBM.

Por último, si modificamos un algoritmo cargado y queremos guardar los cambios, tenemos la opción de guardar. Si el usuario de stereoTuner clica sobre el botón con el icono de una tarjeta de memoria (menú de calibración), el programa modificará el fichero de calibración del

algoritmo que se esté mostrando y reemplazará los valores de las propiedades por los valores actuales de las barras de ajuste.

3.6.5 Logger

En stereoTuner, al igual que en muchas otras aplicaciones, existe un sistema de *backup* de acciones con todo lo que se ha ido haciendo desde el momento en el que se inicia la ejecución de la aplicación. El nombre que recibe este sistema en stereoTuner es *logger*. El *logger* contiene una lista por orden cronológico, en la que el usuario puede consultar que acciones se han ido ejecutando (por él o por la aplicación). Para acceder al *logger*, el usuario clica en la pestaña "Log" del notebook, situado en el bloque inferior derecho de la pantalla principal. En el *logger* (3.4.7) se distinguen dos columnas, la primera informa del instante temporal en el que se realiza la acción y en la segunda columna el mensaje o *log* que relata la acción.

Los mensajes del *logger* tienen una estructura concreta:

[código de acción] Localización de la acción: Cuerpo del mensaje. Solución al problema.

- Código de acción. Existen cuatro códigos distintos que tipifican cada *log*: INFO, DEBUG, WARNING y ERROR. El primero es el más común, indica que es un mensaje informativo, es decir, informa de la ejecución de una acción. El segundo, no lo veremos aparecer a no ser que estemos desarrollando, lo implementé para que el desarrollador pudiera utilizar el *logger* a modo de consola y mostrar mensajes con el contenido deseado al realizar ciertas acciones. El tercero es poco común, sirve para avisar de que la acción no se ha completado satisfactoriamente, pero que el que no se haya hecho, no provocará un fallo de ejecución. El último informa que la acción no se ha podido llevar a cabo por algún motivo, si este mensaje aparece, es porque no es un error que provoque un fallo total, pero podría provocarlo en alguna acción posterior.
- Localización de la acción. Indica el archivo desde el cual proviene la acción.
- Cuerpo del mensaje. Indica la acción.
- Solución al problema. Esta parte del log, sólo se muestra para los tipos WARNING y ERROR. Indica el posible foco o solución del error.

Existe la posibilidad de exportar toda la lista de acciones en un fichero **txt**. Para ello, el usuario clica sobre el último icono de la barra de herramientas (icono de *download*) para abrir la ventana de exportación (3.4.15), introduce el nombre del fichero y clica sobre el botón de aceptar. Si la validación es positiva y no se ha introducido ningún carácter no imprimible, se genera un fichero con la información del *logger* dentro del directorio *logfiles*, que a su vez se encuentra dentro del directorio de ejecución (*exec*).

```
/*********************************************************/
 *STEREOTUNER LOG FILE
 *Name: test2.txt
 *Created: 28/4/2016 21:15:15
/*********************************************************/


21:14:54 [INFO] logger.hpp: Initializing logger.
21:14:54 [INFO] stereoTest.st: Configuration file loaded successfully.
21:14:58 [INFO] CVProcessing.hpp: Set lines property active.
21:14:58 [INFO] CVProcessing.hpp: Image left_lined_img.png created successfully. Check exec project
directory to view it.
21:14:58 [INFO] CVProcessing.hpp: Image right_lined_img.png created successfully. Check exec project
directory to view it.

21:14:59 [INFO] CVProcessing.hpp: Set color property active.
```

Código 14. Ejemplo de fichero de exportación.

3.6.6 Selección disparidad-profundidad

Cuando el puntero del ratón entra en alguna de las imágenes de stereoTuner, se activan unos determinados eventos que capturan la posición del puntero. Con esta posición se obtienen las coordenadas del puntero en la imagen sobre la esté. Estas coordenadas se muestran en el área de

información (3.4.4) junto a otros dos valores, el nivel de gris de ese pixel sobre la imagen de disparidad y un cuarto valor que depende del estado del tercer botón de la barra de herramientas. Cuando el botón está activo, el cuarto valor del área de información corresponde a la disparidad que se ha calculado entre el punto sobre el que está el puntero y su correspondiente en la otra imagen (este valor se obtiene de la imagen de disparidad). Cuando el botón está inactivo, el cuarto valor del área de información muestra la profundidad, es decir, la distancia que separa el punto seleccionado y la lente de la cámara. En este caso, la profundidad se calcula con:

$$profundidad = \frac{1}{disparidad^2} \tag{3.29}$$

Dónde la disparidad es el valor de la disparidad calculada en ese punto (el valor que se mostraría si el botón estuviera activo).

3.6.7 Líneas de rectificación

Otra de las funcionalidades importantes que se activan desde la barra de herramientas, es la de las líneas de rectificación. Si el usuario clica el cuarto botón de la barra de herramientas (el icono de la escuadra) se sobreimpresionan unas líneas horizontales en las dos imágenes estéreo, como se observa en la figura 22. Estas líneas sirven para comprobar si las dos imágenes están rectificadas.

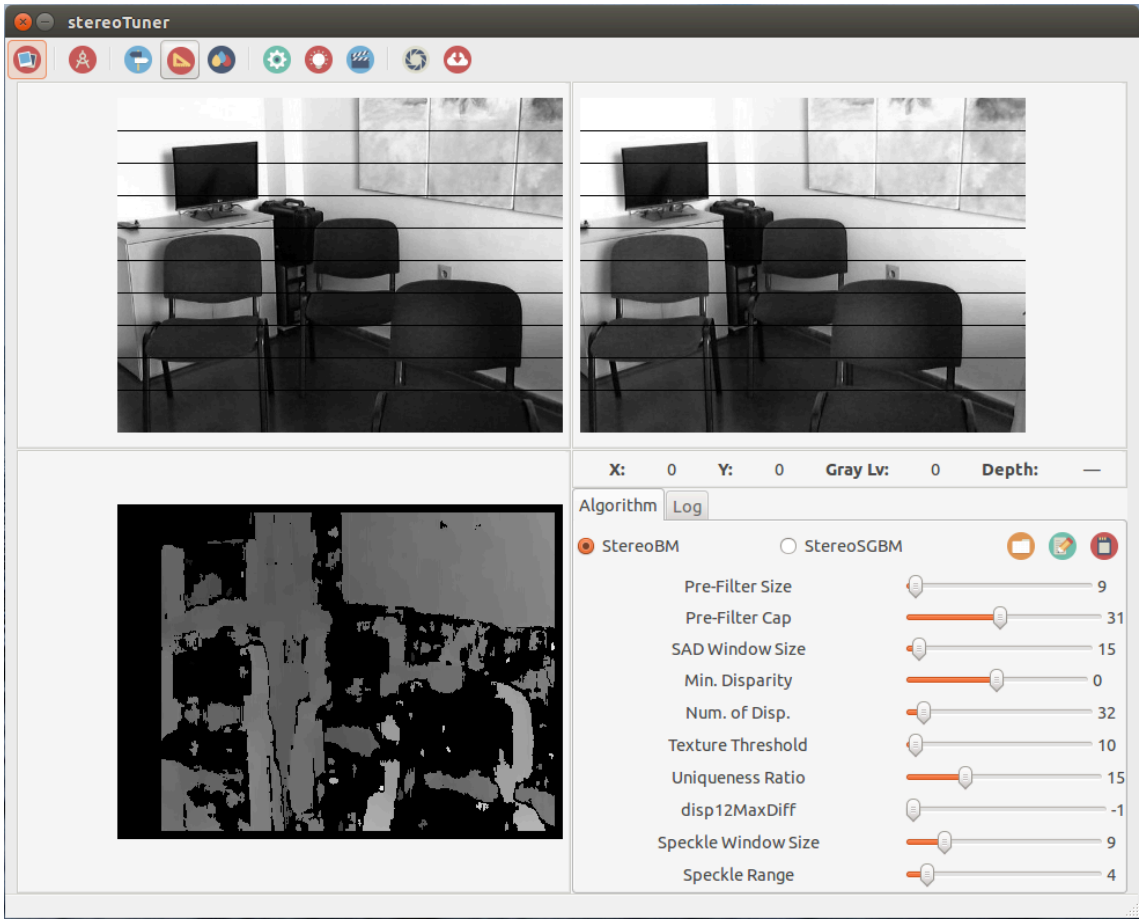


Figura 22. Líneas de rectificación.

Es importante recalcar que las líneas no se añaden a las imágenes reales, ya que una modificación de las imágenes provocaría fallos en el cálculo de la disparidad, además de adulterar la imagen original sin posibilidad de revertir los cambios. Lo que se hace desde el código es duplicar las imágenes y crear dos imágenes nuevas llamadas lined_left.jpg y lined_right.jpg que se crean en el directorio de ejecución (exec). Sobre estas dos nuevas imágenes, se añaden las líneas horizontales. Siempre se colocan 10 líneas, su separación y

posición en la imagen dependerán de sus medidas (anchura y altura). Una vez se han añadido las líneas a las imágenes creadas, se cambia la imagen real por la modificada en la pantalla principal, produciendo el efecto de la sobreimpresión de las líneas en la imagen real. Al desactivar el botón, se vuelven a colocar las imágenes reales en la pantalla principal.

3.6.8 Aplicar mapa de color

Originariamente, la imagen de disparidad se pintaba en escala de grises, haciéndose complicado en ciertas ocasiones, distinguir las pequeñas variaciones en la disparidad. Debido a ello, se añadió la posibilidad de sobreimpresionar un mapa de color sobre la imagen de disparidad. Esta sobreimpresión se lleva a cabo en el mismo punto en el que se calcula la disparidad, en los métodos *st_cvprocessing_compute_stereoBM()* y *st_cvprocessing_compute_stereoSGBM()*. Antes de cargar la imagen en la pantalla principal, se evalúa si la función de aplicar el mapa de color está seleccionada, si no lo está, no realiza ninguna acción y coloca la imagen en escala de grises en la pantalla principal, pero si está activada, obtiene el mapa de color seleccionado del perfil de configuración y lo sobreimpresiona, de manera que los pixeles del mapa de disparidad sigan la escala del mapa de color (figura 23).

Para activar la funcionalidad, el usuario debe clicar el quinto botón de la barra de herramientas. Para modificar el valor de la escala del mapa de color, se debe acceder a la ventana de preferencias (3.4.9) y clicar en el botón que abre el selector de mapas de color (3.4.11), desde esta ventana se selecciona el mapa de color y se clica sobre el botón de aceptar, después se actualizan las preferencias (veremos el proceso en el siguiente punto) y si la funcionalidad está activada, la imagen aparecerá con la nueva escala.

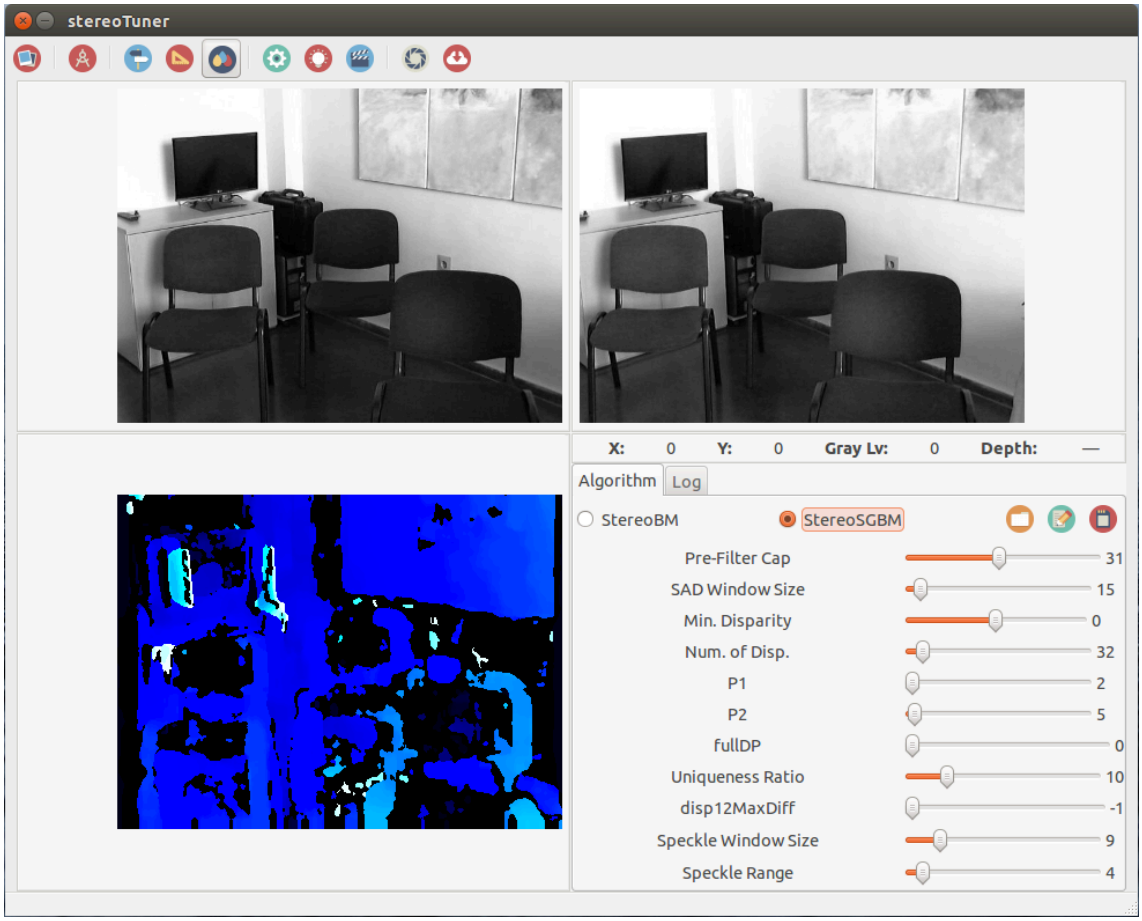


Figura 23. Imagen de disparidad con un mapa de color aplicado.

Se puede escoger de entre 12 mapas de color distintos, estos están extraídos de la librería **OpenCV**. Para aplicar el mapa de color a la imagen de disparidad ya calculada, se utiliza el método de **OpenCV** applyColorMap(). Cada mapa de color tiene un código (un número entero

de entre 0 y 11), se introduce este código y la imagen de disparidad en el método y este devuelve la imagen de disparidad con el mapa de color aplicado. Los mapas de color disponibles en stereoTuner son: *autumn*, *bone*, *cool*, *hot*, *hsv*, *jet*, *ocean*, *pink*, *rainbow*, *spring*, *summer* y *winter* (se pueden observar sus escalas en el directorio ColorMaps, dentro del directorio de ejecución).

3.6.9 Actualización de las preferencias

Cuando se modifican los parámetros del perfil de configuración desde la pantalla principal (activar las líneas de rectificación, seleccionar entre disparidad y profundidad o aplicar mapa de color), realmente no se están actualizando sus valores ni en el **singleton** de las preferencias, ni en el perfil de configuración. Para que se modifiquen y se sobrescriban en el perfil, se tienen que modificar desde la ventana de preferencias (3.4.9).

Cuando desde la ventana de preferencias cambiamos alguna propiedad y clicamos sobre el botón de actualizar preferencias, se sobrescriben todas las propiedades por los valores de la ventana y se realiza un "reinicio total", para que al volver a la pantalla principal todo esté tal y como se haya seleccionado en la ventana de preferencias.

3.6.10 Pantalla de ayuda rápida

La pantalla de ayuda rápida (3.4.13) muestra algunas pistas importantes sobre la aplicación. Yo he implementado dos *tips*: una lista con los formatos de imagen soportados por la aplicación (figura 18) y la secuencia de acciones para llevar a cabo la herramienta Capture.

Los *tips* son imágenes almacenadas en el directorio *tips*, dentro de *resources*, dentro de la carpeta fuente (*src*). Al iniciarse la aplicación, el programa escanea este directorio y carga en el mapa *tips* todos los ficheros que tengan un formato soportado. Cuando se abre la ventana de ayuda rápida, se accede al mapa y se extrae la ruta hacia la primera imagen que es la que se muestra. Como es la primera, el botón de *tip* anterior (icono de una flecha apuntando hacia la izquierda) aparece desactivado y solo se puede clicar en el de *tip* siguiente (flecha apuntando a la derecha) o en el botón de cierre. Cuando se muestra la siguiente imagen (que al ser la segunda, es la última), se desactiva el botón de *tip* siguiente y se activa el de *tip* anterior, de modo que nunca nos salimos del mapa introduciendo un índice que no exista.

Para acceder a la ventana de ayuda rápida, el usuario de stereoTuner tendrá que clicar en el séptimo botón de la barra de herramientas.

Esta funcionalidad está implementada en el código de tal manera, que añadir un nuevo *tip* es muy fácil. Basta con introducir la imagen debidamente escalada en el directorio *tips*, el programa detectará el archivo y lo añadirá al mapa de *tips*. El tamaño de los *tips* ha de ser 600x450.

3.6.11 Pantalla de vídeo de ayuda

La pantalla de vídeo de ayuda, muestra un vídeo en el que se describe el uso de las funciones de la aplicación. Para acceder a esta pantalla, el usuario clica en el octavo botón de la barra de herramientas, cuyo icono es una claqueta de cine. El funcionamiento de la pantalla es similar al de la de ayuda rápida, aunque actualmente sólo hay un vídeo, por lo que los botones de avanzar y retroceder están desactivados.

Funcionalmente, esta pantalla es algo más compleja, debido a que para poder reproducir el vídeo dentro de un objeto **GTK**, ha sido necesario emplear una librería que permitiera esta sincronización (**GTK** +2 no reproduce vídeo). La librería seleccionada ha sido **FFMPEG**, cuyo funcionamiento con **GTK** es eficiente y no da demasiados problemas de sincronización. Otras posibilidades eran: **OGMRip**, **GStreamer**, **LibVLC-gtk**, **Banshee media player**, etc.

3.6.12 Herramienta Capture

La herramienta Capture es una de las funciones más complejas en stereoTuner y en la que más métodos intervienen. El objetivo de esta herramienta es poder medir la disparidad de un punto mediante la supervisión humana, a fin de comprobar si el resultado obtenido mediante el algoritmo seleccionado, es correcto.

Para activar la herramienta Capture el usuario clica sobre el penúltimo botón de la barra de herramientas cuyo icono es un obturador. Al clicar sobre el botón se realizan varias acciones inmediatas que afectan a elementos de la pantalla principal:

- Se activa el botón que alterna entre los valores de disparidad o profundidad, independientemente del estado inicial y se inhabilita para que no se pueda desactivar mientras esté activa la herramienta.
- Se desactiva el botón de las líneas de rectificación y se inhabilita.
- Se desactivan los botones de acceso a la pantalla de preferencias y a la de exportación del logger.
- Se modifican los valores mostrados en el área de información. Durante la ejecución de la herramienta Capture se mostrará: Coordenada en la imagen izquierda, coordenada en la imagen derecha, diferencia y disparidad.
- Se desactivan los eventos en la imagen derecha y en la imagen de disparidad, por lo que si movemos el ratón por alguna de estas imágenes, no se modificarán los valores en el área de información.

Una vez se ha activado la herramienta, el siguiente paso que debe seguir el usuario es el de seleccionar el punto que se quiere estudiar. Para ello tendrá que ir a la imagen izquierda y clicar sobre el punto escogido. Cuando el ratón entre en la imagen izquierda desaparecerá y en su lugar se mostrarán unas líneas de medida en color negro. Estas constan de una línea horizontal a lo largo del ancho de la imagen y una vertical a lo alto, cuyo punto de corte será el punto sobre el que se encuentre el puntero. Conforme nos desplacemos en la imagen izquierda, observaremos que el primer valor del área de información (*Lpx*), irá mostrando las coordenadas del punto sobre el que es encuentre el puntero y el último valor (Disparity) mostrará el valor de la disparidad calculada mediante el algoritmo que esté seleccionado. Para seleccionar el punto, bastará con clicar sobre la imagen, en ese momento, las rayas de medida se quedarán fijas, así como el valor del punto fijado que además, se pintará en rojo (figura 24). Un dato a tener en cuenta es que se puede modificar el color de las barras de medida antes de que cliquemos sobre la imagen izquierda. Si clicamos con el botón derecho del ratón, las barras cambiarán de color. Los colores disponibles son: negro, azul, verde, rojo, azul claro, amarillo, violeta y blanco.

Para mostrar las barras sobre las imágenes, se procede de manera similar a las líneas de rectificación. Como no es apropiado modificar las imágenes originales imprimiendo sobre ellas las barras de medida (producirían fallos en el cálculo de la disparidad), se crea una nueva imagen para cada una de las imágenes del par: measure_left.jpg y measure_right.jpg. El cálculo de la disparidad se realiza sobre las dos imágenes originales, pero cuando se deben mostrar las líneas en las imágenes, se cambia la imagen original por la nueva creada y sobre esta, se imprimen las líneas. Cuando es necesario "borrar" las líneas, se vuelven a poner las originales.

El siguiente paso será desplazarnos a la imagen derecha, que ahora sí, cuando nos movamos sobre ella se mostrarán las barras de medida y el valor de las coordenadas en el segundo valor del área de información (*Rpx*). El tercer valor, mostrará la diferencia entre las coordenadas x en las que estamos situados en la imagen derecha y el punto fijado en la izquierda ($x_{der} - x_{izq}$). Si nos situamos exactamente en el punto de la imagen correspondiente al fijado (la coordenada y será cero si las imágenes están rectificadas), el valor de la coordenada x del tercer valor del área de información (*Diff*), mostrará la disparidad que existe en ese punto. Valiéndonos del valor de la disparidad del cuarto valor del área de información y el valor que acabamos de medir, sabremos si el cálculo matemático de la disparidad difiere mucho del valor medido. Si observamos un valor diferente, podría deberse a que se estén emparejando dos puntos que no

sean correspondientes, pero sí iguales (por lo tanto muy próximos), esto es común en superficies homogéneas como paredes o cuadros.

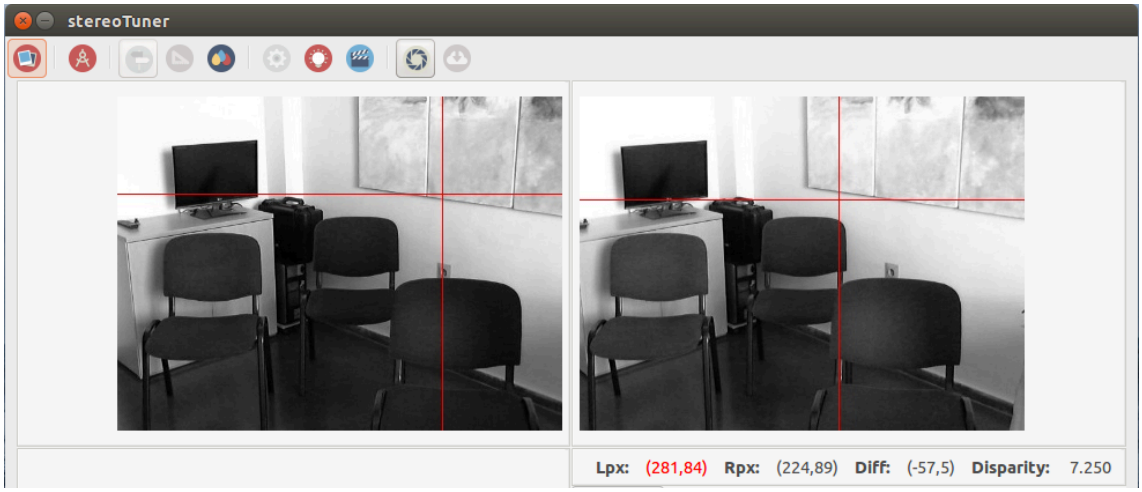


Figura 24. Barras de medida en la herramienta Capture.

Para retornar a la ejecución normal de la aplicación, el usuario desactivará el botón de la herramienta Capture y la pantalla principal volverá al mismo estado en el que estaba antes de activarse la herramienta.

Una última curiosidad acerca de la modificación del puntero. Cada vez que entremos en una imagen y aparezcan las barras de medida, el puntero desaparecerá, pero cada vez que salgamos de la imagen, volverá a aparecer y desaparecerán las barras de medida.

3.6.13 Validaciones en stereoTuner

Como he venido mencionando anteriormente, en stereoTuner se realizan validaciones en todos los selectores de archivos y en todas las entradas de texto, para prevenir futuros errores de ejecución o de lectura. En el caso de los selectores de archivos, es imprescindible que sólo se deje cargar el archivo que se necesite cargar, es decir, si se desea modificar la imagen izquierda, no tendría sentido que dejáramos cargar un perfil de configuración. Por lo tanto, en los selectores de archivos se realizan las siguientes validaciones:

- Selector de imágenes. Se dejan cargar solo los archivos que se encuentren dentro del mapa de formatos soportados (*formats*). Si la validación falla muestra el siguiente mensaje de error: *File extensión not supported. View all supported formats on tips window* (figura 26).
- Selector de perfiles de configuración. Sólo se dejan cargar archivos con extensión st. Si la validación falla se muestra el siguiente mensaje de error: *File extension not supported. Only stereoTuner files (.st)*.
- Selector de ficheros de calibración. Se dejen cargar archivos con extensiones BM y SGBM. En este caso hay un matiz, si el selector se abre desde el menú de calibración, deja cargar cualquiera de los dos archivos soportados y si falla la validación se muestra: *File extensión not supported. Only stereoTuner algorithm files (.BM or .SGBM)*. Pero si se abre desde el menú de preferencias, sólo deja cargar la extensión desde la cual se haya abierto el selector. Si se abre desde "StereoBM:", dejará cargar sólo archivos BM y si falla la validación, mostrará: *File extension not supported. Only BM calibration files*. Si se abre desde "StereoSGBM:", sólo dejará cargar archivos SGBM y si la validación falla, mostrará: *File extension not supported. Only SGBM calibration files*.

En las entradas de texto, lo que se valida es que no se introduzcan caracteres no imprimibles por los principales SO (Windows, Mac OSx y Linux), a fin de evitar que el nombre que le demos al fichero no sea el nombre que acabe teniendo en el sistema o para evitar que se añada alguna extensión al fichero. El método que se ejecuta de manera general cuando se introduce texto en stereoTuner es *validateFilename()*, este método compara cada carácter de la cadena de texto

introducida, con los caracteres “prohibidos” y si encuentra alguna coincidencia devuelve un objeto booleano a *true*, indicando que existe un error en la cadena y que no se puede proseguir. Los caracteres no imprimibles son: ‘ . , * | \ : “ < > ¿ ? / ~ # % & { } ’ (figura 25).

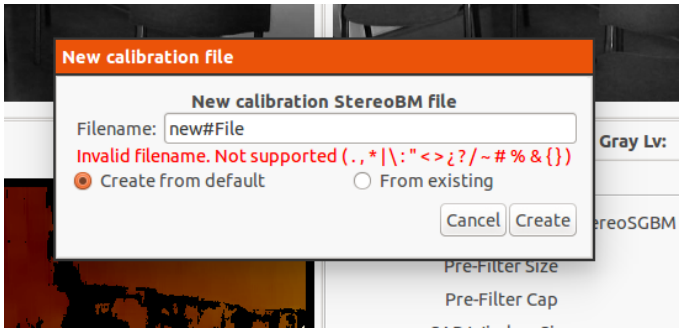


Figura 25. Error de validación en una entrada de texto.

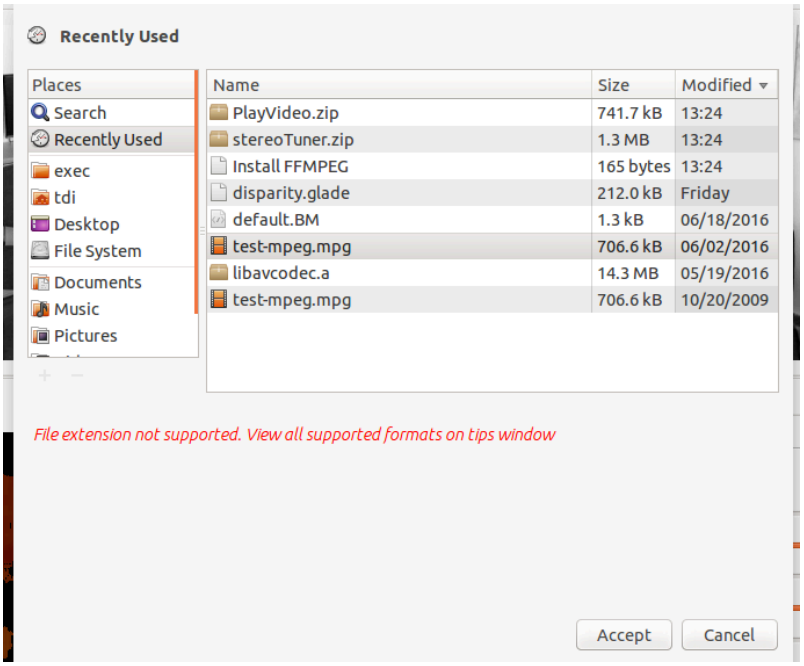


Figura 26. Error de validación al cargar una imagen.

Capítulo 4. Conclusiones y líneas de trabajo futuras

4.1 Conclusiones

Como conclusión al proyecto, me gustaría extender mi impresión acerca del trabajo realizado y de los resultados obtenidos. Durante cerca de 6 meses, he estado trabajando en la aplicación y creo que el resultado final ha superado con creces las expectativas iniciales. Me he sentido muy cómodo con las ideas iniciales y más aún si cabe, con las que he ido proponiendo yo, que además, he podido llevar a cabo de forma creativa. Considero importante el trabajo creativo y el intraemprendimiento dentro de un proyecto, ya que cuantas más ideas nazcan, más crecerá y mejorará el proyecto. En el aspecto estético considero que la mejoría ha sido notoria y que aunque siempre se puede mejorar, estoy orgulloso del trabajo realizado.

Es cierto que durante el desarrollo de la aplicación, me han venido montones de ideas a la cabeza, algunas se han desvanecido ante otras que les ganaban la partida y otras eran demasiado complejas temporalmente para llevarlas a cabo. Muchas de estas ideas, las llegué a desarrollar como objetivos secundarios y finales (2.2 y 2.3) de la aplicación, sin embargo, son estas ideas complejas, las que reservé para futuros desarrollos y por tanto, nombraré en la siguiente sección.

4.2 Líneas futuras

4.2.1 Procesado en tiempo real

Sería interesante que el cálculo del mapa de disparidad se hiciera en tiempo real. Es decir, que se pudiera conectar por USB una cámara estéreo y que en lugar de visualizar las imágenes estéreo estáticas actuales, visualizáramos en tiempo real las imágenes captadas por la cámara. Para hacer esto, tendríamos que realizar determinados cambios en la aplicación.

- Tendríamos que generar un sistema de directorios dentro del directorio src/calibration, en el que se fueran creando directorios por cada cámara distinta que fuéramos utilizando. Por tanto, tendríamos que diseñar un código que escaneara los puertos USB y buscara la cámara estéreo, extrajera su nombre y creara un directorio con el nombre de la cámara (dentro del directorio *calibration)*. Por supuesto, habría que comprobar que no existe un directorio ya creado para esa cámara antes de crearlo.
- Una vez creado el directorio, habría que crear los ficheros de calibración que necesita **OpenCV** para poder rectificar las imágenes (ficheros con los parámetros extrínsecos e intrínsecos de la cámara). En el caso de que ya tuviéramos creado el directorio para esa cámara y dispusiéramos de los ficheros, este paso se saltaría. Para crear los ficheros de calibración, tendríamos que implementar un código para calibrar la cámara en tiempo real. Para ello, necesitaríamos un tablero de ajedrez y llamar a los métodos de **OpenCV** findChessboardCorners() y drawChessboardCorners() en primer lugar, lo siguiente sería llamar a la función stereoCalibrate() para obtener las matrices de rotación y traslación (R y T) y las matrices fundamental y esencial (F y E). El siguiente punto sería llamar a las funciones undistortPoints() y computeCorrespondEpilines(). Con esto ya habríamos

generado los ficheros extrínsecos e intrínsecos de calibración, los cuáles guardaríamos en el directorio de la cámara.

- El siguiente paso, sería rectificar las imágenes que vienen de la cámara. Para poder realizar este paso, necesitamos los ficheros de calibración, de ahí el punto anterior. Hay que mencionar, que el proceso de calibración sólo haría falta ejecutarlo una vez, una vez dispusiéramos de los ficheros de calibración, iríamos directamente a rectificar las imágenes. Para rectificar las imágenes utilizaríamos el método **OpenCV** stereoRectify().

Una vez estuviéramos obteniendo en tiempo real, cada *frame* de la cámara rectificado, lo pasaríamos por los métodos ya desarrollados para calcular el mapa de disparidad (*st_cvprocessing_compute_stereoBM()* y *st_cvprocessing_compute_stereoSGBM()*), que mostrarían el mapa de disparidad en tiempo real.

Este desarrollo es muy interesante para comprobar que conseguimos captar bien la profundidad de los objetos. Si al procesar las imágenes utilizamos uno de los mapas de color de los que dispone la aplicación, podríamos alejar y acercar objetos a la cámara y ver como varían los colores que imprimen estos objetos, a fin de concretar que la aplicación es capaz de calcular su profundidad en la escena.

4.2.2 Uso de Point Cloud Library

Point Cloud Library (PCL) es un proyecto independiente, a gran escala y abierto para el procesamiento de nubes de puntos 2D/3D. La idea para un futuro desarrollo, sería la de mediante el uso de esta librería, utilizar las imágenes captadas en tiempo real para generar una nube de puntos. Con esta nube de puntos, podríamos generar una reconstrucción tridimensional de la escena. Este reconstrucción la podríamos exportar a programas de edición 3D como **Blender** y tratar los objetos a nuestra conveniencia. Es cierto que este desarrollo se sale de la calibración estéreo y entra de lleno en la reconstrucción tridimensional, pero, ¿no es ese el siguiente paso lógico de cara a continuar la aplicación?.

Para entender cómo funciona **PCL** y cómo podríamos generar la nube de puntos a partir de lo existente en la aplicación, hagamos la siguiente comparativa: **PCL** sería como un sensor 3D si entendemos a **OpenCV** como un sensor 2D. Mientras que **OpenCV** procesa objetos Mat (superficies bidimensionales) **PCL** procesa *point clouds* (nubes de puntos). Una superficie bidimensional está compuesta por pixeles (x, y) y una *point cloud* está compuesta de puntos (x, y, z). De esta comparativa es fácil extraer como podríamos generar las nubes de puntos a partir de lo desarrollado en la aplicación. El punto 2D ya lo tenemos, para conseguir la tercera coordenada (z), sólo necesitamos calcular la disparidad en ese punto y a partir del valor de la disparidad obtener la profundidad, para de este modo formar un punto de tres coordenadas, necesario para elaborar la nube de puntos que necesita **PCL** para trabajar.

4.2.3 Uso de contenedores Docker

El concepto básico que hay detrás de **Docker**, es el de crear contenedores portables y ligeros para que una aplicación software pueda ejecutarse en cualquier máquina que tenga **Docker** instalado, independientemente del sistema operativo que dicha máquina tenga por debajo. Para poder acceder a una aplicación, esta necesita estar ejecutándose en una máquina, pero además, necesita una serie de cosas para hacerlo correctamente: versión del código utilizado (c++, java, phyton, etc), librerías con versiones específicas, o cualquier cosa ajena al código desarrollado en sí. Pues bien, **Docker** permite meter en un contendor (entendámoslo como una "caja") el código de la aplicación y todas las cosas que la aplicación necesita para ejecutarse. Así que visto de algún modo, **Docker** es cómo una máquina virtual (Linux) muy ligera, que nos permite crear múltiples sistemas (totalmente asilados entre sí) sobre la misma máquina (ordenador). La gran diferencia con una máquina virtual convencional, es que las máquinas virtuales necesitan contener todo el sistema operativo, mientras que un contenedor **Docker** aprovecha el sistema operativo desde el que se está ejecutando (comparte el mismo kernel e incluso algunas librerías).

Hay dos términos que hay que diferenciar en **Docker**: contendores e imágenes. Las imágenes en **Docker** son componente estáticos, ya que no dejan de ser un sistema operativo con ciertas aplicaciones empaquetadas, mientras que un contenedor es la instanciación de esa imagen, pudiendo ejecutar varios contendores de la misma imagen, de forma paralela. Si hacemos una analogía con el lenguaje orientado a objetos, la imagen equivaldría a una clase y el contenedor a la instanciación de esa clase, es decir, a un objeto.

Mi última propuesta de desarrollo futuro, es la de incorporar **Docker** a la aplicación. Puesto que para poder ejecutar stereoTuner en una máquina, se necesitan ciertas librerías: **GTK** (versión 2.8), **OpenCV** (versión 2.4.10), **FFMPEG** (versión 3.0), **CMake** (versión 2.6) y obviamente, un sistema operativo Linux. El hecho de incorporar **Docker** a la aplicación, haría que pudiéramos ejecutar nuestra aplicación en cualquier máquina con cualquier sistema operativo (**Docker** funciona de manera nativa sólo en Unix, pero mediante la virtualización de boot2docker se puede ejecutar en Windows y OSX) sin necesidad de instalar absolutamente nada. Es decir, cualquier persona que quisiera comenzar a desarrollar en stereoTuner, podría comenzar de inmediato sin necesidad de encontrar las librerías y sus versiones específicas, con el riesgo que a veces supone configurar nuestra máquina con librerías de versiones específicas.

Capítulo 5. Metodología del TFG

5.1 Gestión del proyecto

La primera reunión con el tutor se produjo a principios de 2015, si bien aún no estaba matriculado del trabajo final de grado, buscaba un proyecto que me permitiera ser creativo y original, en el que no se me pusiera límites o restricciones a la hora de aportar mis propias ideas. A lo largo del segundo cuatrimestre de 2015, fuimos reuniéndonos eventualmente a fin de buscar el proyecto idóneo, si bien al principio se pensó en otro desarrollo, a finales de año, ya en el primer cuatrimestre del presente curso (2015/2016), definimos el proyecto y comencé a trabajar en él.

El planteamiento del proyecto ha sido el siguiente:

- Diciembre: definición de objetivos, establecimiento de plazos, lectura de la documentación necesaria para el inicio del proyecto y configuración del entorno de trabajo.
- Enero-Mayo: desarrollo del código de la aplicación. De manera paralela se fue documentando lo realizado.
- Mayo-Junio: redacción del documento y revisión de la aplicación.

5.2 Distribución en tareas

Me propuse un plazo para trabajar en el proyecto de 4/5 meses, dentro de ese intervalo, la propuesta general era desarrollar el máximo número de tareas posibles, por lo que, una vez se finalizaron los objetivos iniciales, se fueron proponiendo nuevos objetivos. No ha sido hasta el inicio de la redacción de este documento, cuando se ha finalizado el desarrollo de la aplicación, que aproximadamente ha llevado 5 meses. La lista de tareas llevadas a cabo ha sido la siguiente:

1. Lectura de documentación y comprensión de las tecnologías y librerías especificas de la aplicación.
2. Comprensión y estudio del código inicial (el esqueleto de la aplicación).
3. Estructuración de los ficheros y directorios, así como la optimización del código inicial.
4. Implementación de la barra de herramientas.
5. Implementación del *notebook*.
6. Implementación del área de información.
7. Estructuración y estilismo de la pantalla principal.
8. Desarrollo del fichero y perfil de configuración: stereoTuner-config.xml y default.st.
9. Desarrollo inicial del *parser*: XMLsT-parser.hpp.
10. Implementación de la configuración inicial de la aplicación a partir de los datos en el fichero de configuración.
11. Implementación de la ventana de preferencias a partir de los datos obtenidos del perfil de configuración.
12. Implementación del selector de archivos y de las diferentes validaciones.

13. Implementación del *logger*.
14. Implementación de la ventana del asistente de perfiles de configuración y ampliación del *parser*.
15. Desarrollo de la funcionalidad de las líneas de rectificación.
16. Desarrollo de la funcionalidad de seleccionar entre disparidad y profundidad.
17. Desarrollo de la funcionalidad de aplicar el mapa de color a la imagen de disparidad.
18. Implementación de la ventana de selección de mapas de color.
19. Desarrollo de la funcionalidad Capture.
20. Implementación de la ventana de exportación del *logger*.
21. Implementación de la ventana de ayuda rápida.
22. Creación de los ficheros de calibración (default.BM y default.SGBM) e implementación de la ventana para su creación.
23. Optimización del código.
24. Redacción de "stereoTuner Reference Manual".
25. Redacción de la memoria.

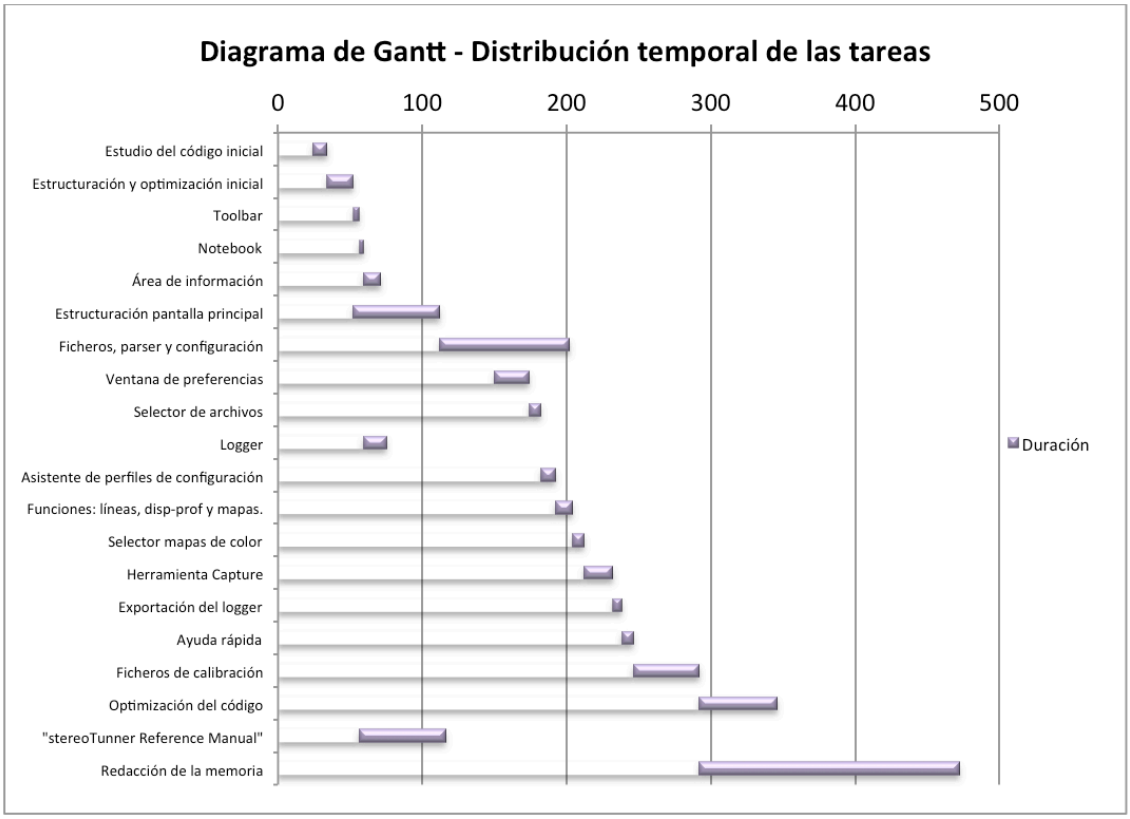
5.3 Diagrama temporal

Para observar mejor la progresión temporal en concepto de horas dedicadas a cada tarea propuesta en la tabla 1, he elaborado un diagrama de Gantt, en función del periodo y la duración de las tareas.

Tarea	Fecha	Coste (h)
1. Lectura de la documentación.	14/12/2015 – 21/12/2015	24
2. Estudio del código inicial.	21/12/2015 – 4/1/2016	10
3. Estructuración y optimización inicial.	21/1/2016 – 4/1/2016	18
4. Toolbar	11/1/2016	4h
5. Notebook	12/1/2016	3h
6. Área de información	13/1/2016 - 28/3/2016	12h
7. Estructuración de la pantalla principal.	11/1/2016 - 26/5/2016	60h
8. Desarrollo de los ficheros de config. 9. Desarrollo inicial del parser. 10. Configuración de la aplicación.	25/1/2016 - 10/5/2016	90h
11. Ventana de preferencias.	29/2/2016 - 7/3/2016	24h
12. Selector de archivos y validaciones.	7/3/2016 - 8/3/2016	8h
13. Logger	12/1/2016 - 14/3/2016	16h
14. Asistente de perfiles de configuración.	14/3/2016 - 16/3/2016	10h
15. Líneas de rectificación. 16. Selección disparidad-profundidad. 17. Aplicación mapas de color.	16/3/2016 - 21/3/2016	12h
18. Selector de mapas de color	21/3/2016 - 22/3/2016	8h
19. Herramienta Capture	23/3/2016 - 28/3/2016	20h
20. Exportación del logger.	29/3/2016 - 30/3/2016	6h
21. Ayuda rápida	18/4/2016 - 20/4/2016	8h

22. Ficheros de calibración	25/4/2016 - 10/5/2016	46h
23. Optimización del código.	12/5/2016 - 26/5/2016	54h
24. “stereoTuner Reference Manual”	11/1/2016 - 13/6/2016	40h
25. Redacción de la memoria.	26/5/2016 - 28/6/2016	180h

Tabla 1. Timeline tareas.



Gráfica 1. Diagrama temporal.

El diagrama temporal contemplado en la tabla 1 y la gráfica 1, es orientativo. Con esto quiero decir, que el número total de horas no es estrictamente el empleado en cada tarea, pero si muy aproximado, puesto que he ido detallando la fecha de inicio y fin de cada tarea. Para determinar las horas empleadas, he hecho una interpolación basándome en dos tipos de jornada de trabajo: de 4 y 12 horas. La de 4 horas, la he aplicado a la mayoría de días laborables, es decir, de Lunes a Viernes. La de 12 horas, la he aplicado a festivos y los fines de semana que trabajé en el proyecto. Por lo tanto, si dividimos las tareas en tres grupos, el coste total de horas ha sido el siguiente:

Grupo	Tareas involucradas	Coste (h)
Lectura, estudio y preparación.	Tareas 1, 2 y 3.	52
Desarrollo de la aplicación.	Tares 4-23.	381
Documentación.	Tareas 24 y 25.	220h

Tabla 2. Coste total.

De la tabla 2, se desprende el coste total en horas (aproximadas) del proyecto, ascendiendo a un total de 653 horas. Si se supusiera una jornada de 8 horas al día y 40 a la semana, el proyecto habría tenido una duración de 16 semanas (82 días). El tiempo real de dedicación al proyecto ha sido de 7 meses (29 semanas), desde Diciembre hasta Julio.

Capítulo 6. Pliego de condiciones

6.1 Condiciones generales

6.1.1 Condiciones generales

Los siguientes puntos son de obligado cumplimiento entre la parte suministradora (alumno) y la parte contratante (universidad).

- El uso indebido del software o modificación del mismo, excluye a la parte suministradora de cualquier incidencia producida por la parte contratante. Se entenderá como uso adecuado, el uso que se ajuste a las directrices recogidas en este documento y al uso unilateral de la documentación técnica redactada por la parte suministradora.
- Queda excluida toda responsabilidad legal, en lo respectivo a incidencias producidas en los desarrollos posteriores del software y/o versionado del mismo.
- El uso y desarrollo de las líneas futuras de este documento queda a disposición total de la parte contratante, siendo decisión suya su implantación en el software.
- Cualquier incidencia asociada al hardware utilizado por la parte contratante, que no se ajuste como mínimo a las condiciones particulares de este documento, quedará excluida y será responsabilidad total de la parte contratante.
- El uso indebido del software o hardware técnico, será responsabilidad de la parte contratante y no de la suministradora, por lo que cualquier queja o petición acerca del mismo, deberá ser remitida a su fabricante particular. Se considerará software técnico todo programa o aplicación independiente a stereoTuner.

6.1.2 Condiciones económicas

- Las condiciones económicas quedan sujetas por los presupuestos elaborados en el presente documento, quedando a elección de la parte contratante el tipo de presupuesto.

6.2 Condiciones particulares

6.2.1 Condiciones hardware

Los siguientes puntos hacen referencia a las necesidades físicas mínimas para el funcionamiento del proyecto, sin ellas no se podría llevar a cabo su desarrollo o funcionamiento.

- Equipo informático: Será necesario un equipo con un mínimo de 2Gb RAM. El sistema operativo deberá ser Linux o OSX. En el caso de necesitar el uso de una máquina virtual, el equipo tendrá que tener un mínimo de 2 núcleos (Dual Core o Core2Duo) para separar los procesos de ejecución, de modo que el rendimiento de la aplicación sea el mínimo requerido.

6.2.2 Condiciones software

Los siguientes puntos hacen referencia al software necesario para el desarrollo del proyecto.

- Entorno de desarrollo: Será necesario un entorno de desarrollo con soporte para los siguientes lenguajes de programación: C++, **xml** y css.
- Entorno gráfico: Es aconsejable el uso de **glade** para la programación gráfica de **GTK+**. Como la versión de **GTK+** utilizada por el software es la 2, las versiones de **glade** soportadas serán: 3.8 (versión estable) o 3.8.5 (actualización).
- Entorno de edición: Será necesario un entorno de edición de imágenes, para llevar a cabo la edición y/o modificación de los iconos mnemónicos de la aplicación, así como cualquier otro aspecto similar que pudiera ser requerido.
- Máquina virtual: En el caso de que el equipo se ejecute en un SO distinto a los requeridos, será imprescindible el uso de una máquina virtual con SO Linux para su ejecución y desarrollo.
- Esquematización y diagramas de flujo: Para llevar a cabo los diagramas y esquemas técnicos, será necesario un software de edición de diagramas de flujo.
- Documentación: Para redactar la documentación técnica, podrá utilizarse cualquier editor de texto que pueda convertirse a PDF.

6.2.3 Condiciones de calidad

El proyecto en general, o cualquier sistema necesario para satisfacer su desarrollo en particular, se diseñará y utilizará siguiendo la normativa estipulada por los principales organismos reguladores (normativa aplicable al proyecto). Estos organismos son: UNE, IEC (*International Electrotechnical Commission*) y ETSI (*European Telecommunications Standards Institute).*

6.2.4 Condiciones de seguridad

El presente proyecto se ajusta a las directrices de la Organización Internacional del Trabajo (OIT) en materia de seguridad y salud laboral, preservando el bienestar social, mental y físico del trabajador. Por lo que cumple la ley 31/1995 de Prevención de Riesgos Laborales desarrollada en el artículo 40.2 de la Constitución.

6.3 Presupuesto y Coste

6.3.1 Aspectos generales

En los siguientes apartados voy a elaborar los presupuestos para llevar a cabo el proyecto y en última instancia el coste real de mi desarrollo. En concepto de software he elaborado dos presupuestos, uno utilizando software libre (*open source*) y otro utilizando software con licencia.

6.3.2 Presupuesto hardware

Elemento	Descripción	Coste
Equipo informático	Ordenador portátil	1.000€

Tabla 3. Presupuesto hardware.

6.3.3 Presupuesto software

Elemento	Descripción	Coste
Entorno de desarrollo	IntelliJ IDEA (JetBrains software)	499€
Entorno gráfico	GLADE (Gnome software)	0€
Entorno de edición	Photoshop Cs6 (Adobe software)	580€
Máquina virtual	VMware Workstation 12 Player (VMware software)	134.95€

Esquematización y diagramas	StarUML (MKLab software)	62.71€
Documentación	Office 2016 (Microsoft Software)	539€

Tabla 4. Presupuesto software con licencia.

Elemento	Descripción	Coste
Entorno de desarrollo	Eclipse (Eclipse Foundation software)	0€
Entorno gráfico	GLADE (Gnome software)	0€
Entorno de edición	Gimp (Gimp org software)	0€
Máquina virtual	VM VirtualBox (Oracle software)	0€
Esquematización y diagramas	StarUML vOS (MKLab software)	0€
Documentación	Libre Office (The document foundation software)	0€

Tabla 5. Presupuesto software libre.

6.3.4 Presupuesto prototipo

Asumiendo que este proyecto se pudiera comercializar, el coste del prototipo sería el reflejado por las tablas 6 y 7 (dependiendo del tipo de software que se utilizara). Para calcular el coste del software desarrollado se ha hecho el cálculo teniendo en cuenta las horas que ha costado desarrollarlo.

Si tenemos en cuenta que el precio medio de un ingeniero *senior* ronda los 60.000€ brutos anuales, lo que serían aproximadamente 30€/hora y el coste de horas del proyecto ha sido de 473 horas (no contamos las 180 horas de la redacción de este documento, para más información ver la tabla 2), desarrollarlo habría costado 14.190€. Por lo que el coste por licencia, presuponiendo que tendría que haber un mantenimiento (para solventar posibles errores en el código) y versionado del software con carácter anual, sería de 299€.

Elemento	Coste
Coste hardware	1.000€
Coste software	0€
Coste software stereoTuner	299€
TOTAL:	**1299€**

Tabla 6. Presupuesto prototipo software libre.

Elemento	Coste
Coste hardware	1.000€
Coste software	1815.66€
Coste software stereoTuner	299€
TOTAL:	**3114.66€**

Tabla 7. Presupuesto prototipo software con licencia.

6.3.5 Coste

En la tabla 8 expreso el coste de los elementos que he utilizado para llevar a cabo el proyecto, añadiendo el coste de mano de obra, equiparando mi hora de trabajo a la de un ingeniero Junior, lo que serían 15€/hora aproximadamente:

Elemento	Descripción	Coste
Equipo informático	Ordendor de mesa Mac OSx	1600€
Entorno de desarrollo	Eclipse (Eclipse Foundation software)	0€
Entorno gráfico	GLADE (Gnome software)	0€
Entorno de edición	Photoshop vStudent (Adobe software)	199€
Máquina virtual	VM VirtualBox (Oracle software)	0€
Esquematización y diagramas	StarUML (MKLab software)	62.71€
Documentación	MacOffice 2016 vStudent (Windows software)	149€
Coste desarrollo	653 horas a 15€/hora	9795€

Tabla 8. Coste proyecto.

El subtotal (coste de software y hardware) es de 2010.71€, el total (subtotal y mano de obra) asciende a **11.805,71€.**

Capítulo 7. Bibliografía

7.1 Documentación técnico-sanitaria

[1] Puell Marín, Dra. Mª Cinta, "Óptica Fisiológica. El sistema óptico del ojo y la visión binocular" *Universidad Complutense de Madrid*.

[2] Aguilar, M.; Mateos, F., "Óptica Fisiológica Tomo I" *Servicio de publicaciones UPV*. Valencia, 1993.

[3] Álvarez, J. L.; Tapias, M., "Tema 1: Generalidades sobre la visión binocular" *Universidad Politécnica de Cataluña.*

[4] Diseñado por Freepik (figura 1). Creative Commons.

7.2 Documentación técnica

[1] Bradski, G.; Kaehler, A., "Learning OpenCV" *O'REILLY*. United States of America, 2008.

[2] Konolige, K., "Small version systems: hardware and implementation" *in English International Symposium on Robotics Research, 1997.*

[3] Hirschmüller, H., "Semi global block matching" *IEEE Transactions on pattern analysis and machine intelligence.* Vol. 30, Nº 2, Febrero 2008.

[4] Hartley, R. I., "Theory and practice of projective rectification" *International Journal of Computer Vision35.* 1998.

[5] Bouguet, J. -Y., "Camera calibration toolbox for Matlab". 2008.

7.3 Documentación tecnológica

[1] GNOME, "GTK+ 2 Reference Manual" http://library.gnome.org/devel/gtk2. [Online].

[2] C++, "C++ Reference" http://www.cplusplus.com/reference. [Online].

[3] GNOME, "Glade User Interface Designer Reference Manual" http://developer.gnome.org/gladeui. [Online].

[4] OpenCV, "OpenCV API Reference" http://docs.opencv.org/2.4.10/modules/refman.html. [Online].

[5] De Román Marínez, A., "stereoTuner Reference Manual", 2016.

[6] FFmpeg, "Documentation" http://ffmpeg.org/documentation. [Online].

[7] CMake, "Reference Documentation" http://cmake.org/documentation/. [Online].

[8] Point Cloud Library, "PCL API Documentation" http://docs.pointclouds.org/trunk. [Online].

[9] Docker, "Dockerfile reference" http://docs.docker.com/engine/reference/builder. [Online].

[10] Meyers, S., “More Effective C++” *Addison-Wesley professional computing series.* 1996.

Capítulo 8. Anexos

8.1 Fundamentos fisiológicos

8.1.1 Cálculo de la posición y el tamaño de la imagen óptica por la aproximación del ojo enfocado al infinito

La imagen de un objeto situado en el infinito se forma en el plano focal imagen y su tamaño depende del ángulo subtendido por el objeto. Un rayo procedente del extremo de un objeto A incide en el punto principal H formando un ángulo u con el eje óptico. El rayo se refracta y desvía hacia el eje, formando un ángulo u' con él. Por trigonometría el tamaño de la imagen y' se obtiene de,

$$\mathbf{y'} = -\boldsymbol{f'} + \mathbf{\tan}\,\boldsymbol{u'} \tag{8.1}$$

En esta expresión se ha introducido el signo negativo para satisfacer el convenio de signos ya que f' y tan u' son positivos pero y' es invertida y debe tener un valor negativo.

Para ángulos pequeños se puede aproximar que sen u ≈ tan u, y de acuerdo con la ley de refracción

$$\mathbf{n}\,\mathbf{\sin}\,\boldsymbol{u} = \boldsymbol{n'}\,\mathbf{\sin}\,\boldsymbol{u'} \tag{8.2}$$

En este caso n es el índice de refracción del aire, y como el ángulo u es muy pequeño la última expresión se puede poner en la forma paraxial más simple

$$\boldsymbol{u'} = \frac{\boldsymbol{u}}{\boldsymbol{n'}} \tag{8.3}$$

Por lo tanto:

$$\boldsymbol{y'} = -\boldsymbol{u'f'} = -\boldsymbol{u}\frac{\boldsymbol{f'}}{\boldsymbol{n'}} \tag{8.4}$$

como

$$\frac{\boldsymbol{n'}}{\boldsymbol{f'}} = \boldsymbol{F_o} \tag{8.5}$$

Tenemos:

$$\boldsymbol{y'} = -\frac{\boldsymbol{u}}{\boldsymbol{F_o}} \tag{8.6}$$

En esta expresión y' está en metro, u en radianes y F en dioptrías, veamos un ejemplo concreto.

Si tenemos un objeto de 3m de altura que es visto por un ojo con una potencia equivalente de +60D a 20 metros de distancia. Su posición y el tamaño de la imagen óptica que forma serán:

$$s' = f' = \frac{n'}{F_o} = \frac{1{,}336}{60} = 0{,}0223\ m = 22{,}3\ mm \tag{8.7}$$

$$\tan u = \frac{3}{20} = 0{,}15 \approx u = 0{,}15\ rad \tag{8.8}$$

$$y' = -\frac{u}{F_o} = -0{,}15\frac{1000}{60} = -2{,}5\ mm \tag{8.9}$$

8.1.2 Diplopía patológica debida al estrabismo

En el estrabismo los dos ejes visuales no se cruzan en el punto objeto de mirada. Se producirá diplopía, a no ser que la imagen del ojo desviado se suprima de manera que no se perciba. En ausencia de supresión, tal visión doble se llama diplopía patológica y se puede representar en el modelo de ojo cíclope.

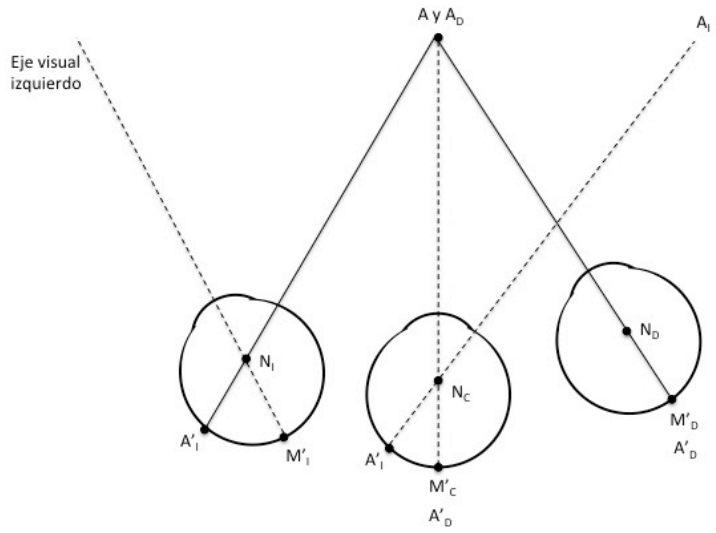


Figura 27. Diplopía patológica a través del ojo cíclope.

En la figura X está representado un estrabismo divergente o exotropía del ojo izquierdo, en el que el ojo derecho fija el punto objeto A. La imagen de A cae en la retina temporal del ojo izquierdo y se proyecta a través del ojo cíclope al lado derecho del punto de fijación, mientras en el ojo derecho la imagen de A (A'$_D$) se proyecta a través del ojo cíclope, para coincidir con el objeto A. La diplopía resultante es cruzada, estando la imagen percibida por el ojo izquierdo (A$_I$) a la derecha de aquella percibida por el ojo derecho (A$_D$).