Universidad Zaragoza

Proyecto Fin de Carrera

Desarrollo de una aplicación software para la fusión robusta de mapas de profundidad y reconstrucción 3D densa

Autor

Sergio Ibáñez Sanahuja

Director

Pedro Piniés Rodríguez

Escuela de Ingeniería y Arquitectura

2013

Dedicado a toda mi familia. Para los que están, y para los que están por venir.

La investigación es para ver lo que todo el mundo ha visto, y pensar en lo que nadie ha pensado.
Albert Szent-Györgi

Agradecimientos

Quisiera expresar mi agradecimiento a todo el mundo que me ha apoyado a lo largo de la realización del proyecto, a mi familia, especialmente a mi madre y mi pareja ya que su apoyo ha sido incondicional, a mis compañeros de clase durante estos maravillosos cinco años, a mis compañeros de piso por agradarme este largo verano especialmente a mi compañero en el día a día Francisco Javier, con el que he compartido un verano de duro trabajo. Por último al director del proyecto, Pedro, por su paciencia y apoyo durante todo el proyecto. Gracias a todos porque sin vuestro apoyo esto no hubiera sido posible.

Índice

Índice de figuras

Resumen

El objetivo principal de este proyecto es mejorar la calidad de los mapas de profundidad que se han obtenido a partir de un conjunto de imágenes en niveles gris capturadas por una cámara monocular. Un mapa de profundidad es simplemente una imagen en la que los píxeles, en vez de almacenar colores, almacenan la profundidad a la que se encuentran los objetos en la escena. El ámbito de aplicación de estos mapas de profundidad es muy variado, reconstrucción 3D, detección de obstáculos (útil por ejemplo para vehículos), algoritmos de tracking de cámara, segmentación de una escena (identificación de distintos objetos que la componen), etc.

La mejora propuesta en este proyecto consiste en en la aplicación de un algoritmo de fusión de mapas de profundidad llamado TGV-fusion, para ello se van a utilizar técnicas variacionales de optimización densas, es decir, tienen en cuenta información en toda la imagen. La principal ventaja de esta técnica es que son idóneas para su programación sobre hardware paralelo, en particular en este trabajo se utiliza la arquitectura CUDA (Compute Unified Device Architecture) desarrollada por NVIDIA que extiende el lenguaje de programación C para codificar algoritmos paralelos en GPUs según el paradigma SIMT (Single Instruction Multiple Thread), es decir, una misma instrucción ejecutada en múltiples threads simultáneamente.

Debido a la complejidad matemática de los algoritmos involucrados y a las dificultades iniciales propias de la falta de experiencia programando en CUDA, se tomó la decisión en los inicios del proyecto, de resolver primero el problema de image denoising (eliminación de ruido en imágenes) debido a que el algoritmo de esta solución está muy relacionado con el de la fusión de mapas pero es más sencillo de resolver e interpretar de forma intuitiva.

En cuanto a los resultados generados en este proyecto son, en primer lugar la explicación intuitiva de los métodos matemáticos necesarios, definición del algoritmo de denoising para distintas normas, implementación de una aplicación para el estudio del efecto de los parámetros en este algoritmo, aplicación en tiempo real y aplicación final del TGV-fusión.

Capítulo 1

Introducción

La obtención automática de modelos 3D densos de un determinado entorno usando una cámara como único sensor tiene un gran interés debido a su bajo coste y al amplio abanico de aplicaciones: reconstrucción y modelado de obras artísticas, obtención de modelos densos de ciudades (Google Maps), asistencia en cirugías no invasivas como laparoscopias u otras aplicaciones que no dependen estrictamente de la reconstrucción como en la industria del entretenimiento donde se puede usar para desarrollar juegos para consolas Xbox o Wii que permiten interactuar al jugador con el mundo real usando realidad aumentada, detección de obstáculos en navegación de vehículos, segmentación, etc.

El proceso para obtener una reconstrucción densa a partir de un conjunto de imágenes es laborioso y complejo. Desde un punto de vista de alto nivel podemos identificar distintos bloques, a continuación se explica todo este proceso.

1.1. Contexto en el que se enmarca el proyecto

En la figura 1.1 se muestran los distintos bloques en los se divide el proceso desde el cálculo de los mapas de profundidad hasta las distintas aplicaciones que pudieran hacer uso de ellos. A continuación se detallan de forma resumida estos bloques.

1.1.1. Localización de la cámara

La clave para poder conseguir extraer un depth map[1] de cualquier escena es conseguir información de la profundidad, para ello se requiere más información que una sola imagen.

[1]Depth map: Mapa de profundidad

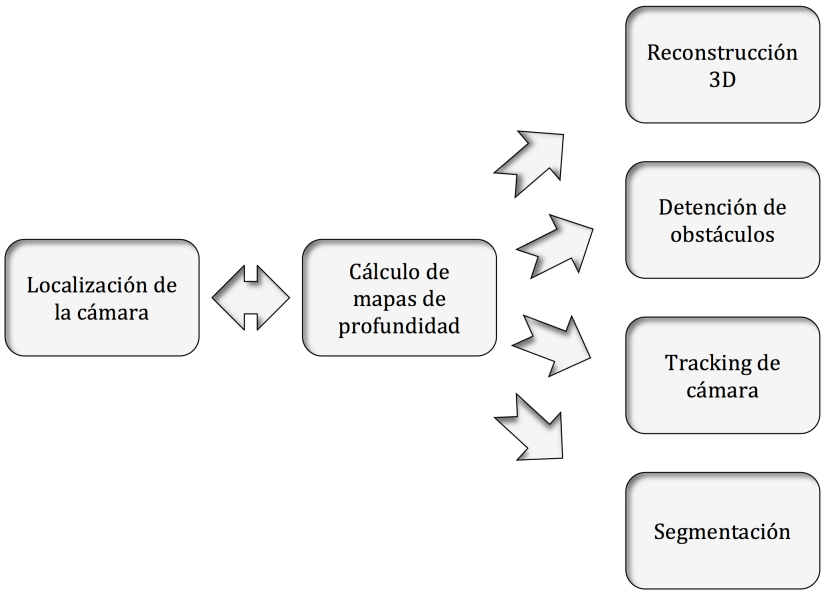


Figura 1.1: Esquema de todo el proceso desde la consecución de los mapas de profundidad hasta hasta las aplicaciones de este

Los mapas de profundidad usados como entrada del algoritmo implementado en este proyecto, se han construido a partir de información capturada por una cámara monocular. Puesto que de una sola imagen no es posible extraer información de la profundidad, como si que sería posible en una cámara estéreo, se requiere información redundante para conocerla. Esta se consigue al tomar varias imágenes y conocer con precisión el desplazamiento de la cámara al tomar estas, de este modo se podrá deducir la profundidad a la que se encuentra el punto en cuestión. Este proceso también se puede realizar a la inversa, lograr a partir un mapa de profundidad la localización espacial de la cámara en el momento de realizar las distintas capturas.

1.1.2. Mapas de profundidad

En todos los aspectos, un mapa de profundidad es equivalente a una imagen, sólo que en vez de almacenar valores de color, almacena distancias.

A partir de un conjunto de imágenes en niveles de gris y las correspondientes posiciones de la cámara desde donde se tomaron se puede crear un mapa de profundidad mediante el algoritmo *Dense Tracking and Mapping*[RAND11]. La idea intuitiva es que si la profundidad de un punto en el espacio 3D está bien calculada, los niveles de gris de los píxeles proyectados en todas las imágenes en las que se vio debería coincidir. Desafortunadamente esta técnica produce imágenes de profun-

didad ruidosas y con espurios que pueden afectar a la calidad de las aplicaciones que hace uso de ellas. El objetivo del este proyecto es eliminar o al menos reducir estos efectos en los depth maps construidos como se explica en el apartado 1.3.

1.2. Estado del arte

En cuanto al estado del arte las investigaciones en las que se basa este trabajo son relativamente recientes.

Por un lado artículos sobre sistemas que se encargan de todos los procesos necesarios para obtener una reconstrucción 3D usando una cámara monocular, es decir abarcan todo el proceso mostrado en la figura 1.1, por ejemplo *Dense Tracking and Mapping in Real-Time* [RAND11] o *Dense Reconstruction On–the–Fly* [AWB]. Además se encargan de realizarlo en tiempo real.

Existe también otros artículos que tratan este problema pero usando cámaras Kinect que dan mucha información ya que consiguen de manera sencilla cada uno de las mapas de profundidad, por ejemplo Kinect-Fusion [SI11].

Por último otro artículo muy relacionado es el *TGV-Fusion* [TPB11], pero con la diferencia de que este se centra en la reconstrucción a partir de imágenes aéreas, por lo que parte del proceso realizado en este proyecto no es necesaria. En este caso las imágenes de entrada van a ser imágenes de corta distancia donde la escena cambia mucho con un leve movimiento del punto vista, por lo que será necesaria la programación de un mecanismo de virtualización como se explicará más adelante.

Todos estos trabajos son la fuente de multitud de aplicaciones tales como las comentadas al inicio del capítulo.

1.3. Proceso propuesto

La aportación propuesta en este proyecto con respecto al esquema anterior aparece en la figura 1.2. En concreto se ha incluido un módulo que es le que se va a encargar que realizar la fusión.

Este módulo se basa en el artículo de TGV-fusion [TPB11] para imágenes aéreas pero con la particularidad de que se realizará un proceso intermedio para la transformación de los mapas de entrada por lo dicho en el apartado anterior.

1.3.1. Fusión de mapas

Como se observa en la figura 1.2, para realizar la fusión será necesario más de un depth map, el nuevo bloque se encarga de fusionar estos mapas de profundidad que presenten cierto solapamiento (es decir que todos los mapas tengan una misma región de la escena en común) para generar uno nuevo de mayor calidad.

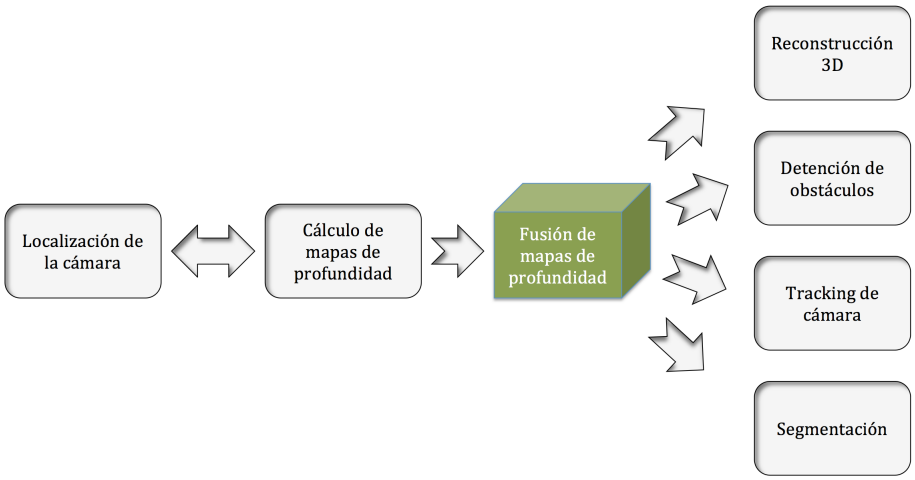


Figura 1.2: Nuevo esquema del proceso desde la consecución de los mapas de profundidad hasta hasta las aplicaciones de este pasando por la fusión

Los mapas de profundidad resultantes de la etapa anterior a este nuevo módulo tienen información redundante, ya que se han generado varios con distintas imágenes, este hecho da la capacidad de fusionar estos mapas para lograr uno de mayor calidad que los anteriores.

Esta fusión se realiza aplicando técnicas variacionales de optimización convexa. En este punto se centra el proyecto, concretamente en la aplicación de este algoritmo basado en Primal Dual para la fusión de estos mapas de profundidad de forma eficiente y robusta.

Como se indicó en el resumen de la memoria, en una primera etapa del proyecto se estudiará el problema del denosising, es decir la eliminación de ruido de una imagen, ya que es un problema bastante similar al de la fusión, ambos tienen la misma base, por lo que una vez programadas todas las normas[2] del denoising la programación de la fusión será más rápida ya que se utilizarán exactamente los mismos conceptos.

1.4. Algoritmo Primal Dual

La optimización de funciones es una rama muy importante de estudio en las matemáticas, en general, no es posible encontrar el mínimo o el máximo absoluto de un problema de optimización aunque sí existen, para determinados tipos de funciones, soluciones óptimas o buenas heurísticas [NY04].

[2]Normas: Cada una de las variantes del algoritmo de optimización utilizado en el denoisisg que se estudiarán más adelante.

Afortunadamente en este proyecto nos encontramos en uno de esos casos, en los que las funciones a minimizar que nos vamos a encontrar, presentan una estructura especial que permite obtener la solución global (óptima) del problema. La estructura genérica de los problemas que vamos a resolver, tanto en denoising como en TGV-fusion, es la siguiente:

$$\min_u F(Ku) + G(u) \tag{1.1}$$

Donde F y G son funciones convexas aunque no tienen por que ser diferenciables y K es un operador lineal. La solución a esta optimización se basa en el algoritmo Primal Dual que como veremos más adelante, cuando se aplica a problemas de visión por computador, se puede acelerar considerablemnte usando tarjetas gráficas (GPUs) y programación paralela en CUDA [CUD10].

1.5. Organización

A lo largo de memoria se van a ir abarcando conceptos de manera gradual para ayudar a la comprensión. Todo el proceso que se sigue a continuación se basa en el algoritmo Primal dual cuyo análisis está realizado en el anexo A.Una vez comprendido este algoritmo se abarcarán los distintos problemas de denoising que se trata, como se ha dicho, de una aplicación muy directa e intuitiva del algoritmo Primal Dual (Capítulo 2), a continuación se abarcará el problema de la fusión así como todos los pasos previos para realizarla (Capítulo 3), por último habrá un pequeño apartado de conclusiones de todo el proyecto donde se expondrán las impresiones personales, y se explicarán posibles líneas futuras de trabajo en relación al proyecto (Capítulo 4).

Capítulo 2

Image Denoising

Como se ha comentado, las funciones de la forma 1.1 se pueden aplicar a muchos problemas, relacionados o no con el tema de este proyecto. Una aplicación casi directa, y muy intuitiva del problema es el uso de esta para eliminación de ruido en imágenes. En este capítulo, se van a detallar tres algoritmos con la forma 1.1 que nos darán pie a, en primer lugar conocer en mas detalle el algoritmo de optimización Primal Dual ya que se trata de una aplicación mas sencilla que la fusión de mapas, y además se aprenderán conceptos muy útiles para apartados posteriores.

2.1. Introducción

Todos los problemas que se van a tratar en este capítulo tienen prácticamente la siguiente estructura:

$$\min_{u} \int |\nabla u| + \frac{\lambda}{2} \parallel u - g \parallel_2^2 d(\Omega) \tag{2.1}$$

Esta esta integral afecta al espacio continuo de la imagen, y como tal se trata de un problema variacional [LIRF92]. Esta compuesta de dos funciones, para lograr la minimización habrá que encontrar un resultado óptimo para ambas partes. Cada una de estas partes tiene un significado valora unas características concretas que deberá tener la imagen resultado.

- $|\nabla u|$: Se trata del regularizador, en este caso es el valor absoluto del gradiente de la imagen propio del método TV-ROF que se verá mas adelante. Este término tratará de valorar soluciones cuya variación entre píxeles sea pequeña, para poder eliminar el ruido de alta frecuencia.

Figura 2.1: Ejemplo de una mala elección de λ, a la izquierda se ha escogido un λ demasiado pequeño, a la derecha uno demasiado grande, y en el centro uno que podría ser óptimo

- $\frac{\lambda}{2} \parallel u - g \parallel_2^2$: Esta parte se denomina data term. Será común a todos los métodos estudiados en el denoising. Valora el hecho de que la solución se parezca lo máximo a la imagen ruidosa. En este caso en particular se trata una norma cuadrática. Existen otras normas que se podrían colocar en este punto tales como la norma L_1 consistente en una función valor absoluto o la norma Huber que será utilizada para la fusión y se explicará más adelante.

En resumen, una colaboración entre ambas eliminará detalles no deseados, mientras que preservará los detalles importantes tales como bordes

El parámetro λ será el que pondere la relación entre ambos términos, por lo que un λ demasiado grande no eliminará el ruido por completo, mientras que uno demasiado pequeño lo hará en exceso eliminando excesiva información. En la figura 2.1 se observa este efecto.

Como la expresión 2.1 es continua, será necesario discretizarla, resultando la siguiente expresión.

$$\min_u F(Ku) + G(u) \tag{2.2}$$

Donde K es un operador discreto equivalente al gradiente, en este caso se trata del cálculo hacia delante del mismo. Es decir, para cada uno de los píxeles calcula la diferencia entre este y el siguiente en filas y en columnas.

$$Ku_{i,j} = (u_{i+1,j} - u_{i,j}, u_{i,j+1} - u_{i,j})$$

La expresión 2.2 se va a sustituir ahora por una equivalente con que la poder trabajar, ya que las funciones $F(Ku)$ y $G(u)$ no tienen porque ser diferenciables, por lo que no es posible la aplicación directa de métodos clásicos. Para ello se realiza una transformación que logra el propósito de convertir la expresión en diferenciable, pero no a cualquier precio ya que la minimización inicial se ha convertido en una maximización y minimización simultáneas.

En resumen, este proceso realiza una dualización de la función no diferenciable, para transformarla a diferenciable.

Todo el proceso detallado para alcanzar la siguiente expresión a partir de la 2.2 se detalla en el anexo A.

$$\min_u \sup_p < Ku, p > +G(u) - F^*(p) \tag{2.3}$$

$$\min_u \sup_{p \text{ s.t. } |p| \leq 1} < Ku, p > +G(u) \tag{2.4}$$

$$\min_u \sup_p C(u, p) \tag{2.5}$$

En primer lugar la expresión 2.3 es directamente la que resulta en el anexo A. En la 2.4, se encuentra la misma pero indicando la función dual como restricción de la maximización. Por último la expresión 2.5 es una simplificación de las dos anteriores que facilitará las operaciones que vienen a continuación.

El algoritmo que se usará para la resolución del problema tiene la siguiente estructura básica.

$$\begin{cases} y^{n+1} = (I + \sigma \partial F^*)^{-1}(y^n + \sigma K \bar{x}^n) & (1) \\ x^{n+1} = (I + \tau \partial G)^{-1}(x^n - \tau K^* y^{n+1}) & (2) \\ \bar{x}^{n+1} = x^{n+1} + \theta(x^{n+1} - x^n) & (3) \end{cases} \tag{2.6}$$

Se trata de un algoritmo llamado *proximal map*, muy similar al gradiente ascendente/descendente pero con unas peculiaridades que lo hace más eficientes.

Se resumen a continuación las operaciones implicadas en cada uno de los pasos.

1. Se trata de la aplicación directa del método del gradiente ascendente a la expresión 2.4, con la particularidad que se realiza la evaluación de y (p en este caso) en el paso siguiente, y que se aplica el operador *proximal map*, para tener en cuenta la restricción que genera la función dual.

2. Esta parte es incluso más parecida a un gradiente descendente clásico ya que para ella no existe restricción.

3. Esta parte es un factor de suavizado que se aplica a la solución para lograr una convergencia mas rápida.

Para conocer más detalles de este método de optimización así como saber en detalle las operaciones del proceso, diríjase al anexo A.

2.2. TV-ROF

En primer lugar se va a estudiar el algoritmo *TV (Total Variation)*, este es el más sencillo de los tres, más directo, y también el primero que se implementó. Se basa en el principio de que las señales con muchos detalles y ruido de alta frecuencia, tienen una gran variación total, es decir, la integral del gradiente en valor absoluto de la señal tiene un valor muy alto. De acuerdo con este principio, la reducción de la variación total de la señal teniendo en cuenta que la señal objetivo se ha de parecer a la original, elimina detalles no deseados, mientras que preserva los detalles importantes tales como bordes.

En este caso la función regularizadora es la función valor absoluto, como en el ejemplo de la introducción del capítulo.

$$F(Ku) = |Ku|$$

En cuanto a su función dual, que será útil para los cálculos posteriores, se encuentra desarrollada en el anexo A.

2.2.1. Proceso

A continuación se van a desarrollar las distintas operaciones que se van a llevar a cabo en cada iteración del método. Para ello,se va a partir de la siguiente expresión, que se simplifica en una función de energía standard como se muestra para entender mejor los cálculos posteriores.

$$\min_{u} \sup_{p \text{ s.t.} |p| \leq 1} < Ku, p > + G(u) \tag{2.7}$$

$$\Downarrow$$

$$\min_{n} \sup_{p} C(u, p)$$

Dual

En primer lugar se esta buscando maximizar $C(u, p)$, por lo que se aplicará el gradiente ascendente.

$$p^{n+1} = p^n + \sigma \nabla_p C(u^n, p^{n+1})$$

Sustituyendo $C(u, p)$ por la ecuación original.

$$\nabla_p C(u^n, p^{n+1}) = Ku^n$$

En este punto se aplica el operador *proximal map* que realiza la proyección del resultado para cumplir la restricción de la expresión 2.7.

$$p^{n+1} = \frac{\tilde{p}^{n+1}}{\text{máx}(1, \tilde{p}^{n+1})}$$

Siendo $\tilde{p}^{n+1} = p^n + \sigma K u^n$.

Primal

Ahora se procede al cálculo del primal, es decir del mínimo de la expresión inicial. Por lo que se aplica en este caso el gradiente descendente.

$$u^{n+1} = u^n - \tau \nabla_u C(u^{n+1}, p^{n+1})$$

Del mismo modo se sustituye $\nabla_u C(u^{n+1}, p^{n+1}) = K^T p^{n+1} + \lambda(u^{n+1} - g)$ con lo que resulta lo siguiente:

$$u^{n+1} = \underbrace{(u^n - \tau K^T p^{n+1})}_{\tilde{u}} - \tau\lambda u^{n+1} + \tau\lambda g$$

Como en el caso anterior se toma una $\tilde{u}$ para simplificar el proceso. Despejando de la expresión anterior u^{n+1} resulta lo siguiente.

$$u^{n+1} = \frac{\tilde{u} + \tau\lambda g}{1 + \tau\lambda}$$

En este caso el operador *proximal map* no realiza ninguna operación puesto que no existe ninguna restricción.

2.3. Huber-ROF

En método de Variación Total tal y como se ha explicado en el apartado anterior sufre del conocido como *Staircasing problem*, esto supone que en las superficies con una leve variación de color tiende a generar pequeñas regiones planas de distintos colores como si de una escalera de distintos tonos se tratase. En la imagen 2.2 se observa este efecto.

El método de Huber intenta poner solución a este problema cambiando la función del regularizador valor absoluto del TV por la norma de Huber.

Esta norma de Huber se basa en una mezcla entre la función cuadrática para valores cercanos a cero y la función valor absoluto para el resto del espectro. Esto hace que para valores pequeños haya una regularización cuadrática, pero para valores grandes una regulación de variación total.

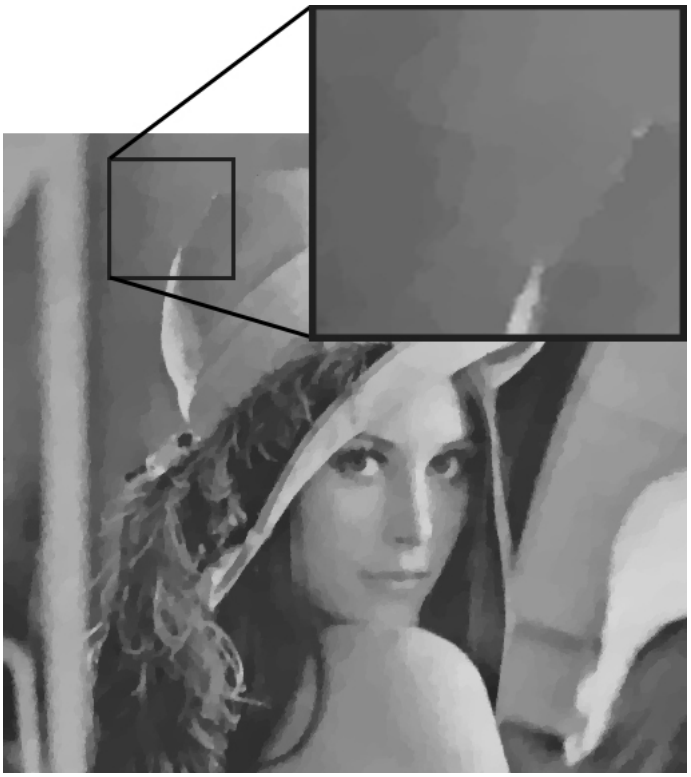


Figura 2.2: Ejemplo representativo de Staircasing Problem

2.3.1. Proceso

Se puede definir la función de la norma de Huber como una función por partes de la siguiente forma.

$$F(x) = \begin{cases} \frac{|x|^2}{2\alpha} & |x| \leq \alpha \\ |x| - \frac{\alpha}{2} & |x| > \alpha \end{cases} \tag{2.8}$$

Cuya función dual es la siguiente.

$$F^*(p) = \begin{cases} \frac{\alpha p^2}{2} & |p| \leq 1 \\ \infty & |p| > 1 \end{cases} \tag{2.9}$$

Teniendo en cuenta las expresiones anteriores, la función a minimizar es la siguiente.

$$\min_u \sup_{p \text{ s.t. } |p| \leq 1} < Ku, p > + G(u) - F^*(p)$$

En este caso $F^*(p)$ no es 0 en los puntos que no hay restricción como el caso del TV, por lo que se sustituye por el valor de la expresión 2.9.

$$\min_{u} \sup_{p \text{ s.t.} |p| \leq 1} < Ku, p > + G(u) - \frac{\alpha p^2}{2}$$

Ademas tiene, por la misma razón del valor absoluto, la restricción de que $|p| \leq 1$.

Las operaciones que definen el bucle principal del método se demuestran de forma similar a como se hacia en el TV, en particular el caso del primal es exactamente el mismo ya que para hacerlo se deriva respecto a u, y como $\frac{\alpha p^2}{2}$ no depende de ella, desaparece. Por ello nos centramos en el cálculo del Dual que es donde afecta el cambio en la función regularizadora.

Dual

Como se hizo para el TV vamos a partir de una función de energía standard $C(u, p)$ y del mismo modo el resultado para la función general del gradiente ascendente es:

$$p^{n+1} = p^n + \sigma \nabla_p C(u^n, p^{n+1}) \tag{2.10}$$

Derivando $C(u, p)$.

$$\nabla_p C(u^n, p^{n+1}) = Ku^n - \alpha p^{n+1}$$

Sustituyendo en la expresión 2.10.

$$\tilde{p^{n+1}} = p^n + \sigma K u^n - \sigma \alpha p^{n+1}$$

Se despeja de p^{n+1} de esta expresión, lo que será la $\tilde{p}^{n+1}$ del *proximal map.*

$$\tilde{p}^{n+1} = \frac{p^n}{1 + \tau\alpha}$$

Ahora se aplica el operador *proximal map* para tener en cuenta la restricción del dual como en el caso del TV.

$$p^{n+1} = \frac{\frac{\tilde{p}^{n+1}}{1+\tau\alpha}}{\max(1, \frac{\tilde{p}^{n+1}}{1+\tau\alpha})}$$

2.4. TGV-ROF

En último lugar se va a detallar el método TGV, es el mas complejo de los tres, más costoso, pero también de los más efectivos, y ademas es la base del método que se usará para la fusión de mapas de profundidad que será descrita en el siguiente

capítulo. Este método parte también con el objetivo de eliminar el *Staircasing Problem* del TV.

En el anexo A se detalla como actúa esta norma sobre la imagen, pero en resumen en superficies planas de la imagen en las que la variación de color sea lineal no habrá penalizaciones que le obliguen a generar pequeñas zonas de igual color como en el TV, si no que permitirá una variación gradual del color.

2.4.1. Proceso

La función del regularizador en este caso será la siguiente.

$$|\nabla u|_{TGV} = \alpha_1|\nabla u - v| + \alpha_2|\nabla v| \tag{2.11}$$

El principal cambio respecto de los métodos anteriores es que añade la variable v y en consecuencia la variable utilizada para indicar su gradiente q, por lo que habrá que hacer el primal y el dual de dos elementos en vez de uno. A pesar de ello el cálculo de las funciones dual es sencillo, ya que se trata de la función valor absoluto en ambos casos.

Se parte pues de la siguiente ecuación para resolver el problema.

$$\min_{u,v} \alpha_1|\nabla u - v| + \alpha_2|\nabla v| + \lambda \parallel u - g \parallel_2^2$$

La expresión que resulta tras aplicar el dual y con la que se partirá en el apartado siguiente se muestra a continuación. Como en otras ocasiones se presentan la misma ecuación de tres formas distintas.

$$\min_{u,v} \sup_{p,q} < K_u u - v, p > + < K_v v, q > + G(u) - F^*(p) - F^*(q) \tag{2.12}$$

$$\min_{u,v} \sup_{p,q \text{ s.t. } |p|\leq\alpha_1;|q|\leq\alpha_2} < K_u u - v, p > + < K_v v, q > + \lambda \parallel u - g \parallel_2^2 \tag{2.13}$$

$$\min_{u,v} \sup_{p,q} C(u, v, p, q) \tag{2.14}$$

Dual

Aplicando el gradiente ascendente resulta la siguiente expresión:

$$p^{n+1} = p^n + \sigma \nabla_p C(u^n, v^n, p^{n+1}, q^n)$$

Derivando la función de energía.

$$\nabla_p C(u^n, v^n, p^{n+1}, q^n) = K_u u - v$$

Se aplica el *proximal map*, para asegurar la restricción en p.

$$p^{n+1} = \frac{\tilde{p}^{n+1}}{\text{máx}(1, \frac{|\tilde{p}^{n+1}|}{\alpha_1})}$$

Ahora se realiza la misma operación pero para q.

$$q^{n+1} = q^n + \sigma \nabla_q C(u^n, v^n, p^n, q^{n+1})$$

Derivando.

$$\nabla_p C(u^n, v^n, p^n, q^{n+1}) = K_v v$$

Aplicando el *proximal map*.

$$q^{n+1} = \frac{\tilde{q}^{n+1}}{\text{máx}(1, \frac{|\tilde{p}^{n+1}|}{\alpha_2})}$$

Primal

Como siempre en primer se calcula el gradiente descendente. Pero esta vez para u y v.

$$u^{n+1} = u^n - \tau \nabla_u C(u^{n+1}, v^{n+1}, p^{n+1}, q^{n+1})$$

$$v^{n+1} = v^n - \tau \nabla_v C(u^{n+1}, v^{n+1}, p^{n+1}, q^{n+1})$$

Derivando.

$$\nabla_u C(u^{n+1}, v^{n+1}, p^{n+1}, q^{n+1}) = K^T p^{n+1} + 2\lambda(u - g)$$

$$\nabla_v C(u^{n+1}, v^{n+1}, p^{n+1}, q^{n+1}) = K_v^T q^{n+1} - p^{n+1}$$

Como en ocasiones sólo será necesario despejar para lograr el resultado.

$$u^{n+1} = \underbrace{(u^n - \tau K^T p^{n+1})}_{\tilde{u}} - 2\tau\lambda u^{n+1} + 2\tau\lambda g$$

$$u^{n+1} = \frac{\tilde{u} + 2\tau\lambda g}{1 + 2\tau\lambda}$$

Y para v.

$$v^{n+1} = v^n - \tau(K_v^T q^{n+1} - p^{n+1})$$

2.5. Elección de parámetros y resultados

La solución que se obtiene para cada uno de los métodos depende de dos tipos de parámetros. Parámetros de convergencia τ y σ (pasos del *proximal map*), que controlan la velocidad de convergencia de la solución. Parámetros de ponderación (λ, α, α_1 y α_2) que controlan el comportamiento entre ajuste a los datos originales y la regularización.

En la literatura sobre Primal-Dual no hay una solución clara en la elección de estos parámetros, por lo que en este proyecto se han desarrollado una serie de aplicaciones que permiten tener una idea de su valor de forma razonable. (Estos valores dependen de la imagen sobre la que actúa el método pero se va a suponer que los parámetros óptimos obtenidos de estas pruebas son extrapolables a cualquier imagen)

En cuanto a las pruebas realizadas a lo largo de todo el capítulo se han desarrollado sobre una tarjeta gráfica *nVidia Tesla M2090*.

2.5.1. Parámetros para la convergencia

Para asegurar la convergencia los pasos τ y σ del *proximal map* deben cumplir la siguiente expresión.

$$\tau\sigma L^2 \leq 1 \tag{2.15}$$

Donde $L = \| K \|$ siendo $\| K \| = \sqrt{8}$, este valor es independiente de la imagen. Se puede encontrar una explicación más detallada de esta elección en el anexo A sección 4.1.

Para buscar los valores τ y σ que mejoran la velocidad de convergencia, es decir para reducir el número de iteraciones para lograr la solución óptima, se ha desarrollado el siguiente proceso.

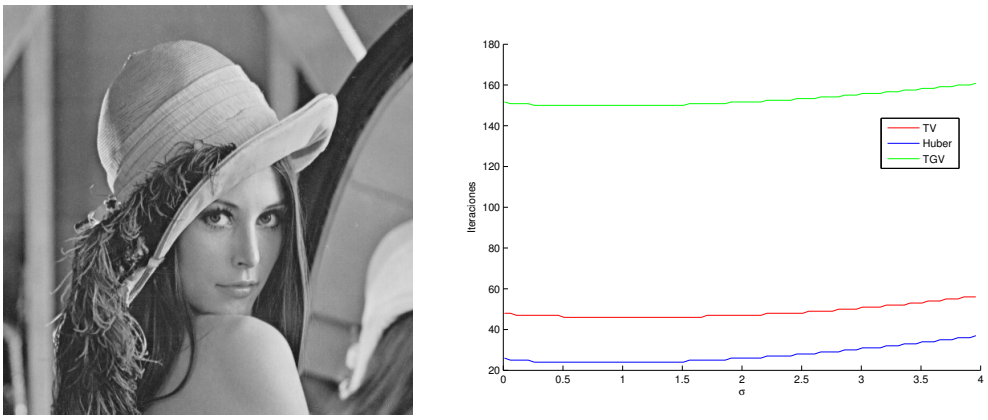
- Se eligen τ y σ tal que se cumple la ecuación 2.15 y como se trata de un problema convexo simplemente se sobreitera el método para llegar aún resultado óptimo u^*.

  Esta solución será utilizada como criterio de parada.

- El criterio de parada se basa en la siguiente ecuación.

$$e_k = \sqrt{\sum_{0}^{nPixels} (u_{ij}^k - u_{ij}^*)^2}$$

  Donde k es la iteración actual. Se ha establecido un umbral para cuando $e \leq 1e-3$ el algoritmo se detenga.



(a) Imagen de la prueba (b) Resultados

Figura 2.3: Resultados generados por la aplicación de prueba de parámetros para σ

El resultado de esta prueba para cada uno de los métodos estudiados se observa en la figura 2.3, así como con la imagen que se ha utilizado para conseguirlo.

El valor óptimo de σ que hacen mínimo el numero de iteraciones es cualquiera entre 0,5 y 1,25 para todos los métodos, se puede obtener el valor de τ simplemente igualando a 1 la ecuación 2.15.

Tiempo de ejecución

Para los valores óptimos de τ y σ elegidos en el apartado anterior, se quiere comprobar el tiempo de ejecución para ver si es posible la ejecución en tiempo real, ya que este depende de las iteraciones necesarias para converger.

TV

Iteraciones	50	100	150	200	250
Tiempo (ms)	11	17	25	31	38

Huber

Iteraciones	25	50	100	150	200	250
Tiempo (ms)	7	10	16	23	30	35

TGV

Iteraciones	150	200	250
Tiempo (ms)	59	76	94

Se puede observar que para TV y Huber es posible aumentar el numero de iteraciones hasta 200, y seguiría ejecutándose en tiempo real, si es que se trabaja a 30

frames por segundo. Para TGV el minimo numero de iteraciones para que converja es 150, es decir 59 milisegundos. Si se fuese más flexible en el tiempo real suponiendo que cada 100 ms fuera posible actualizar la imagen, se podría aumentar hasta 250 el número de iteraciones y conseguir la ejecución en tiempo real.

Se ha implementado una una aplicación que realiza este procedimiento, en el anexo D hay más información.

2.5.2. Parámetros de ponderación

Para elegir los mejores parámetros de ponderación λ(TV, Huber, TGV), α_1 y α_2 (TGV) que regulan el compromiso entre la proximidad de la solución a la imagen de entrada, y regularización del resultado, se ha diseñado el siguiente procedimiento.

1. Para cada algoritmo se realiza el número de iteraciones calculado en la sección 2.5.1, es decir TV 50, Huber 25 y TGV 150.

2. Se obtiene para un rango de los parámetros a estudio, por ejemplo $\lambda_k \in [\lambda_0 ... \lambda_n]$, la solución u^k tras correr el algoritmo el número de iteraciones comentado en el punto anterior.

   Puesto que se dispone de la imagen original sin ruido u^{GT} (Ground truth) que aparece en la figura 2.3 a, se calcula la relación señal ruido SNR entre la solución u^k y la imagen sin ruido u^{GT} mediante la siguiente fórmula.

$$SNR^k = 10 \log_{10} \frac{\sum_{i,j} (u^{GT}_{i,j})^2}{\sum_{i,j} (u^k_{i,j} - u^{GT}_{i,j})^2}$$

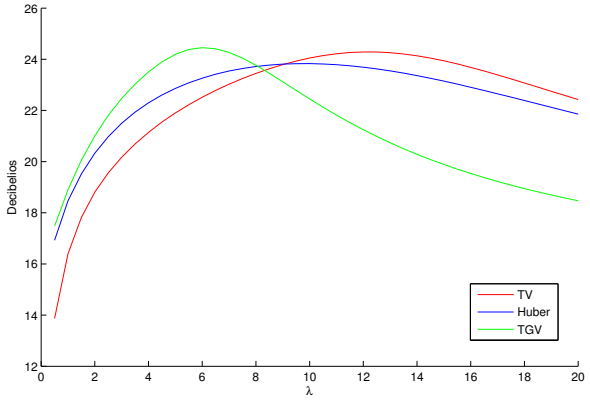
   Se elige el parámetro que maximiza la SNR.

Elección de λ

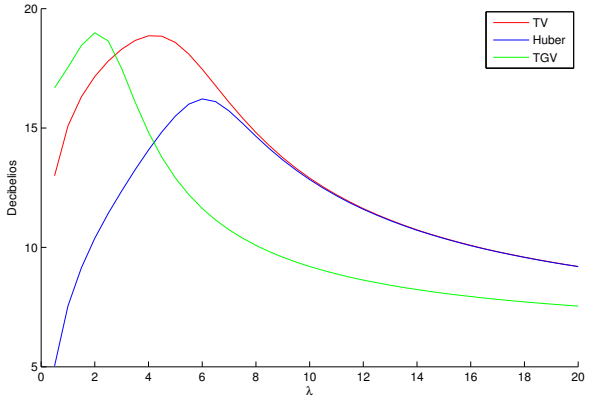
Tanto TV como Huber como TGV dependen de este parámetro. Se recuerda que λ pondera la importancia de la señal de entrada (λ altos) frente a la regularización (λ bajos). Para el caso del TGV que depende de dos valores adicionales (α_1 y α_2) se han fijado a $\alpha_1 = 0{,}5$ y $\alpha_2 = 1{,}5$.

Para realizar este estudio se toma la imagen u^{GT} y se añade ruido gausiano de media 0 y $\sigma = 0{,}1$ (Se recuerda que la imagen u^{GT} esta normalizada entre 0 y 1).

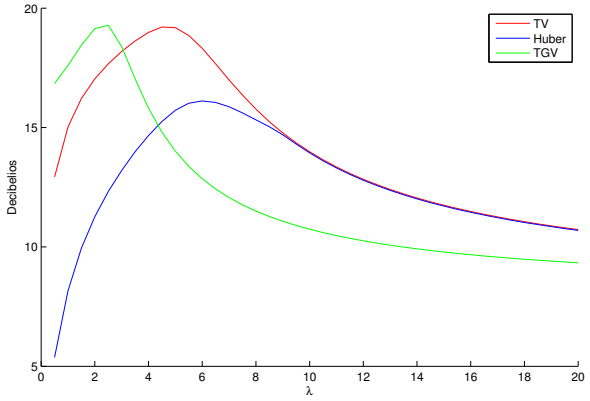
Aplicando el proceso explicado al inicio de la sección se obtiene la gráfica de la figura 2.4 a. Para el TV el mejor es 12, para Huber 8 y para TGV 5. La SNR para la imagen con ruido ($\sigma = 0{,}1$) y la SNR para cada una de las normas utilizando el valor óptimo de λ se puede observar en la figura 2.5.



(a) Ruido gausiano con $\sigma = 0{,}1$



(b) Ruido gausiano con $\sigma = 0{,}3$



(c) Ruido sal y pimienta en el 15 % de la imagen

Figura 2.4: Valores óptimos de λ para distintos tipos y valores de ruido.

(a) Imagen ruidosa $snr = 14{,}4$ (b) TV $snr = 24{,}4$

(c) Huber $snr = 23{,}7$ (d) TGV $snr = 24{,}3$

Figura 2.5: Prueba de comportamiento para los tres métodos estudiados con un ruido gausiano con $\sigma = 0{,}1$

(a) Imagen ruidosa $snr = 6$ (b) TV $snr = 18{,}7$

(c) Huber $snr = 16{,}6$ (d) TGV $snr = 19$

Figura 2.6: Prueba de comportamiento para los tres métodos estudiados con un ruido gausiano con $\sigma = 0{,}3$

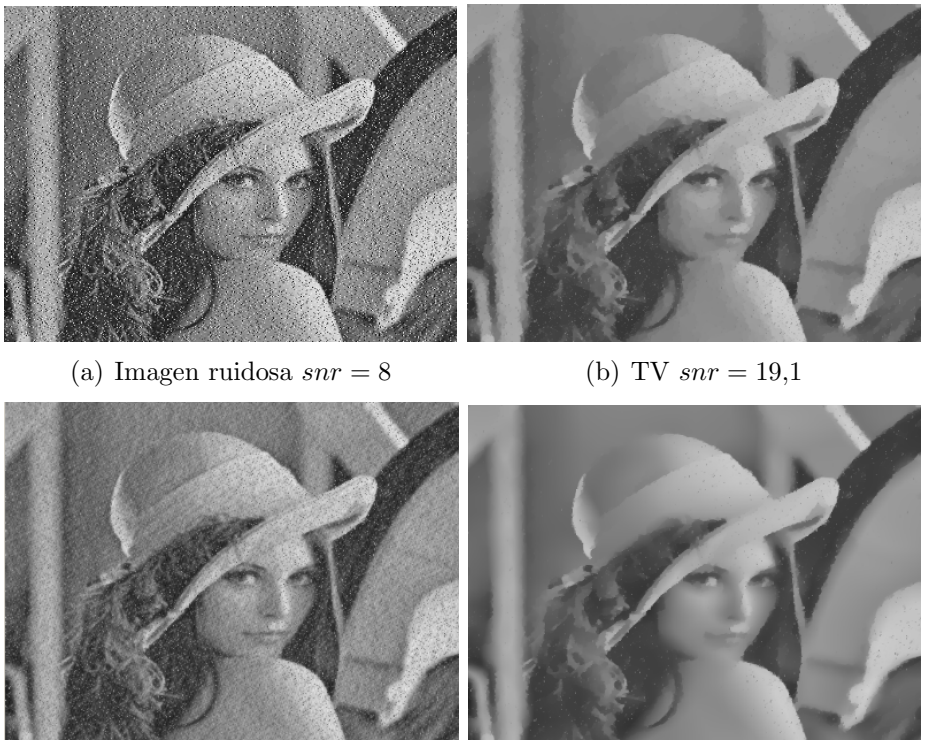
Se observa que todas las normas realizan un gran trabajo. A pesar de que la norma TV logra una mejor SNR, el resultado de TGV es visualmente mejor ya que no se observa *starcasing problem*.

Se estudia también el efecto de subir la desviación estándar del ruido a $\sigma = 0{,}3$. Los valores óptimos de λ para este caso se muestran en la figura 2.4 b, y son para TV es 6, para Huber 4, y para TGV 2. La SNR de la imagen con ruido y para cada norma con los valores óptimos de λ en este caso se observan en la figura 2.6.

Para este caso se observa que a pesar de que exista mejora entre la imagen con ruido y los resultados, estos no son ni de lejos tan buenos como en el caso anterior.

Por último se estudia el efecto del ruido de sal y pimienta. En la figura 2.4 c se muestran los resultados obtenidos para λ para cada una de las normas. Se observa que para TV es 6, para Huber 5 y para TGV 2. Se muestran en la figura 2.7 los resultados obtenidos de esta prueba.

En este caso, como en los anteriores la mejor opción es TGV, pero aun así, nunca



(a) Imagen ruidosa $snr = 8$ (b) TV $snr = 19{,}1$

(c) Huber $snr = 16{,}1$ (d) TGV $snr = 19{,}3$

Figura 2.7: Prueba de comportamiento para los tres métodos estudiados con un ruido sal y pimienta que afecta al 15 % de los píxeles

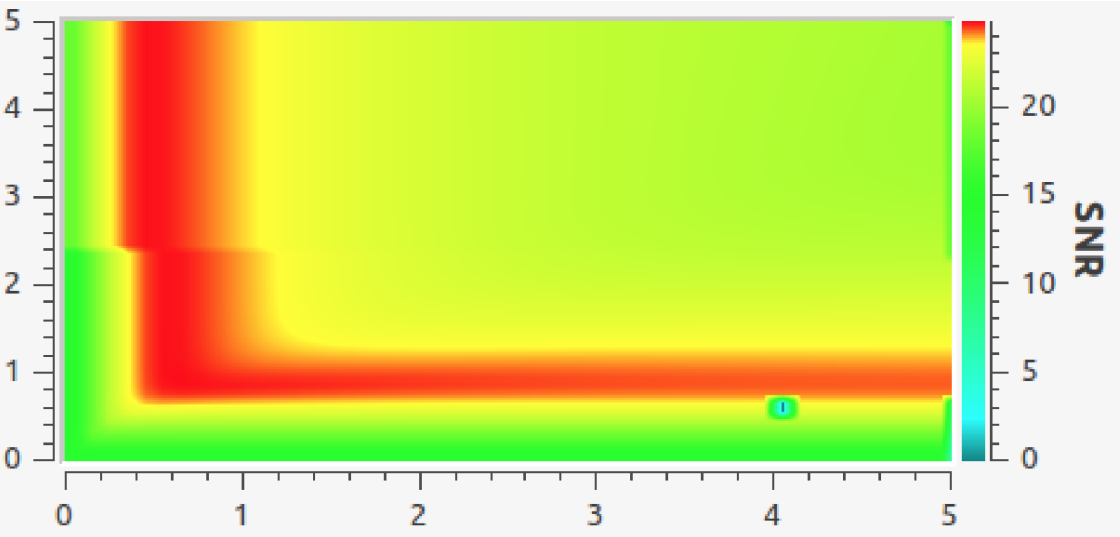


Figura 2.8: Espectrograma resultante del análisis señal ruido de α_1 y α_2, ambas entre 0 y 5, para $\lambda = 5$

llega a suprimir por completo el ruido sal y pimienta, además elimina muchos de los detalles de la imagen al intentarlo.

La norma idónea para este tipo de ruido hubiera sido la TV-L1, aunque para este proyecto no se ha estudiado.

Elección de α_1 y α_2

Para el caso del TGV, para un ruido gausiano de $\sigma = 0{,}1$ y para el valor óptimo de λ calculado en el apartado anterior ($\lambda = 5$), se estudia el efecto de modificar α_1 y α_2. El resultado es el que aparece en la figura 2.8. Se observa que existe un amplio rango de valores de α_1 y α_2 para los que la señal ruido es alta (franja roja), incluidos los valores $\alpha_1 = 0{,}5$ y $\alpha_2 = 1{,}5$ utilizados para las pruebas de la elección de λ.

Para el estudio de todos estos parámetros se ha implementado una aplicación cuyos detalles se encuentran en el anexo D.

Capítulo 3

Fusión de mapas

La parte central del proyecto es la aplicación de los conceptos explicados anteriormente basados en Primal Dual para la fusión de mapas de profundidad.

La base de todo el proceso que se va a explicar a continuación son los mapas de profundidad. Más concretamente un conjunto de ellos, sobre los que se realizarán distintas operaciones para lograr una mejora en la calidad del mapa resultante.

3.1. Mapas de profundidad

Como se ha dicho en la introducción del capítulo, los mapas de profundidad son el principal componente de todo este proceso. Un mapa de profundidad es simplemente una imagen que contiene información relacionada con la distancia de las superficies de los objetos de la escena desde un solo punto de vista. Estos mapas se pueden construir de mediante distintas técnicas, una de las más actuales y que más se están utilizando es el sistema Kinect, que captura las distancias a la escena que tiene delante gracias a un sensor infrarrojo que hay junto a la cámara.

Por otro lado se pueden utilizar otras técnicas para la construcción de mapas de profundidad a partir de imágenes tomadas mediante una cámara monocular como puede ser *DTAM (Dense Tracking and Mapping in Real-Time)* [RAND11]. En este caso se ha utilizado una técnica similar a esta para el cálculo de los mapas.

3.2. Mapas virtuales

Para realizar la fusión de distintos mapas de profundidad deben de estar generados desde el mismo punto de vista. Para ello se ha de realizar un proceso que se detalla en la figura 3.1.

En resumen, lo que hace el sistema es proyectar los datos contenidos en el mapa de profundidad que se quiere trasladar en un cubo 3D, y leer esos datos de vuelta

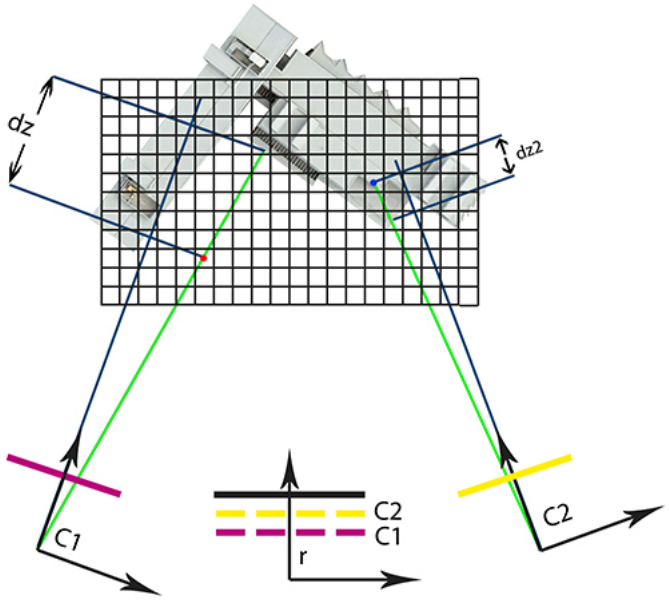


Figura 3.1: Esquema de actuación para conseguir mapas equivalentes

desde la perspectiva de la cámara de referencia r a la que se quiere trasladar el mapa equivalente. El proceso para conseguirlo es el siguiente:

Para cada voxel del cubo se calculará la distancia dz marcada en la figura 3.1. Para ello se traza un rayo desde el origen de la cámara pasando por el voxel hasta su intersección con un objeto de la escena. La distancia del mapa al punto de intersección del rayo con la escena está almacenada en el propio mapa, por lo que es conocida.

Para calcular pues la distancia dz solo será necesario restar la distancia del mapa al objeto intersectado (almacenada en el propio mapa) menos la distancia del mapa al voxel en cuestión.

Observando la figura 3.1, se puede ver que la distancia dz de la que se ha hablado será negativa para zonas que queden detrás de una superficie en la escena (dz_2 en punto azul), y positivas para las zonas visibles (punto rojo).

Una vez relleno el cubo con el depth map que se quiere trasladar solo será necesario leer los vóxeles desde la referencia objetivo r. Para ello, se trazarán rayos para cada uno de los píxeles del mapa virtual contra el cubo. Para conocer el valor de cada píxel del mapa equivalente simplemente hay que buscar el momento en el que el valor del voxel atravesado por el rayo cambia de signo. En ese momento se conocerá a que profundidad z se encuentra la superficie buscada.

Todo este proceso conlleva multitud de operaciones que están detalladas en el anexo E.

3.3. Fusión

Una vez obtenidos todos los Depth Maps, se han de fusionar para lograr uno de mayor calidad que cada uno de ellos por separado. Para ello se va a utilizar una variante del método TGV explicado en el capítulo anterior para el denoising.

La principal razón para que se utilice este algoritmo y no otro es, como se ha recalcado en el capítulo anterior, su buen resultado ante el *Staircasing problem*, que cuando se habla de escenas 3D tiene mas importancia si cabe. Ademas permite aproximar mejor superficies afines en la escena.

Para el data term se utilizará la norma de Huber y no la cuadrática, que se ha utilizado a lo largo de todos los problemas de denoising estudiados en el proyecto, debido a que es mas robusta a la presencia de espurios.

Esta decisión se basa en dos problemas de la norma *ROF* (cuadrática), el comportamiento ante el *Staircasing problem*, y que no es una óptima elección para el ruido esperado, ya que el ruido en estas condiciones es una mezcla entre ruido Gaussiano y valores atípicos, para el que la norma de Huber se comporta mejor.

3.3.1. Proceso

El algoritmo utilizado en este caso es similar al de denoising, pero con mas variables implicadas como se ve a continuación.

La ecuación a minimizar en este caso es la siguiente:

$$\min_{u,v} \left\{ \alpha_1 \int |\nabla u - v| d\Omega + \alpha_2 \int |\nabla v| d\Omega + \sum_{l=1}^{N} \int |u - f_l|_\delta d\Omega \right\} \tag{3.1}$$

En ella se observa por un lado el regularizador, que es similar al de TGV del denoising.

La novedad de esta ecuación se encuentra en el data term, que se ha convertido en un sumatorio, ya que entran en juego más de una imagen, o mapa de profundidad en este caso.

Con la expresión 3.1 no se puede tratar computacionalmente, ya que está definida en el espacio continuo, por lo que es necesario discretizarla. Como en el caso del denoising se dualizan las funciones conflictivas para realizar el método *proximal map*. Cabe destacar la aparición de una nueva variable dual r. Existirá una r_l para cada uno de los mapas de profundidad de entrada con $l \in [1..k]$. Esta se corresponde a la aplicación de la transformada de *Lengendre-Fenchel* al data term.

$$\min_{u,v} \max_{p,q,r} \left\{ \langle K_u u - v, p \rangle + \langle K_v v, q \rangle + \sum_{l=1}^{N} \langle u - f_l, r_l \rangle - \frac{\delta}{2} \parallel r_l \parallel^2 \right\} \tag{3.2}$$

Sujeto a:

$$|p| \leq \alpha_1, |q| \leq \alpha_0, |r_l| \leq 1$$

Como en el caso del denoising se aplican las restricciones correspondientes a las variables duales que han aparecido al realizar la transformación.

A continuación se va a proceder a la descomposición de la expresión 3.2 en cada una de las operaciones que conformarán el método *proximal map*.

Primal

Para el cálculo del primal se deriva en u y en v y se aplica el gradiente descendente:

$$\frac{\partial}{\partial u} = K_u^T p^{n+1} + \sum_{l=1}^{N} (r_l)^{n+1}$$

$$u^{n+1} = u^n - \tau(K_u^T p^{n+1} + \sum_{l=1}^{N} (r_l)^{n+1})$$

$$\frac{\partial}{\partial v} = K_v^T q^{n+1} - p^{n+1}$$

$$v^{n+1} = v^n - \tau(K_v^T q^{n+1} - p^{n+1})$$

Dual

En el caso de las variables p y q se realizan exactamente las mismas operaciones que el caso del TGV del denoising.

$$\tilde{p}^{n+1} = p^n + \sigma(K_u u - v)$$

$$p^{n+1} = \frac{\tilde{p}^{n+1}}{\text{máx}(1, \frac{|\tilde{p}^{n+1}|}{\alpha_1})}$$

$$\tilde{q}^{n+1} = q^n + \sigma(K_v v)$$

$$q^{n+1} = \frac{\tilde{q}^{n+1}}{\text{máx}(1, \frac{|\tilde{p}^{n+1}|}{\alpha_2})}$$

La novedad de este método está en el cálculo de la actualización de las r_l, las nuevas variables que entran en juego.

Para ello, como con todas las anteriores, se calcula el gradiente descendente y se proyecta la solución para cumplir la restricción que impone el dual.

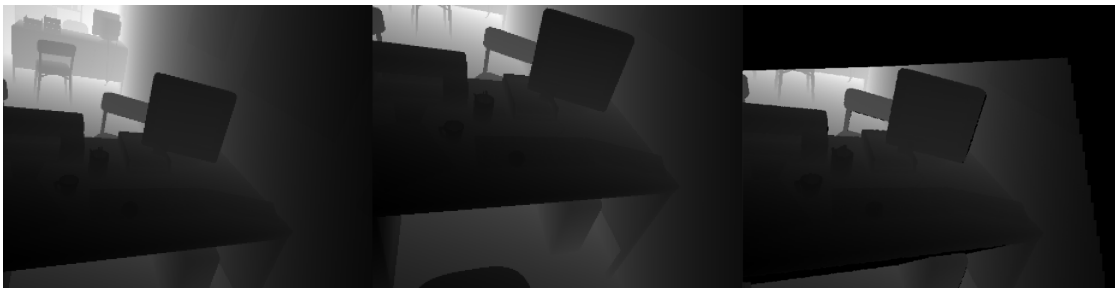


Figura 3.2: Prueba del correcto funcionamiento de la virtualización de mapas de profundidad. Mapa original capturado en la referencia centrada r(Izquierda). Mapa que se quiere transladar a r (Centro). Mapa resultante, ya en r (Derecha)

$$r^{n+1} = r^n + \sigma((u^n - f_l) - \delta r^{n+1})$$

Si se toma $\tilde{r}^{n+1} = r^n + \sigma(u^n - f_l)$, al aplicar la proyección resulta la siguiente.

$$r^{n+1} = \frac{\tilde{r}^{n+1}/(1+\sigma\delta)}{max(1, |\tilde{r}^{n+1}/(1+\sigma\delta)|)}$$

3.4. Resultados de la fusión

Se van a realizar distintas pruebas para comprobar el correcto funcionamiento, tanto de la fusión como del proceso para generar los mapas virtuales.

3.4.1. Generación de mapas virtuales

Para demostrar el correcto funcionamiento de la virtualización de los mapas de profundidad simplemente se va a realizar el cambio de la referencia de un mapa para ponerlo en referencia a otro más centrado.

En la figura 3.2 se observan, por un lado el mapa original capturado desde la referencia objetivo r, en el centro de la misma el mapa original capturado desde su referencia inicial, y por último el mapa virtualizado. Las zonas negras generadas corresponden a zonas no visibles desde la referencia inicial.

3.4.2. Fusión

En primer lugar, antes de demostrar el funcionamiento de todo el proceso, se comprobará que realmente funciona la fusión, para lo que se realizará la siguiente prueba.

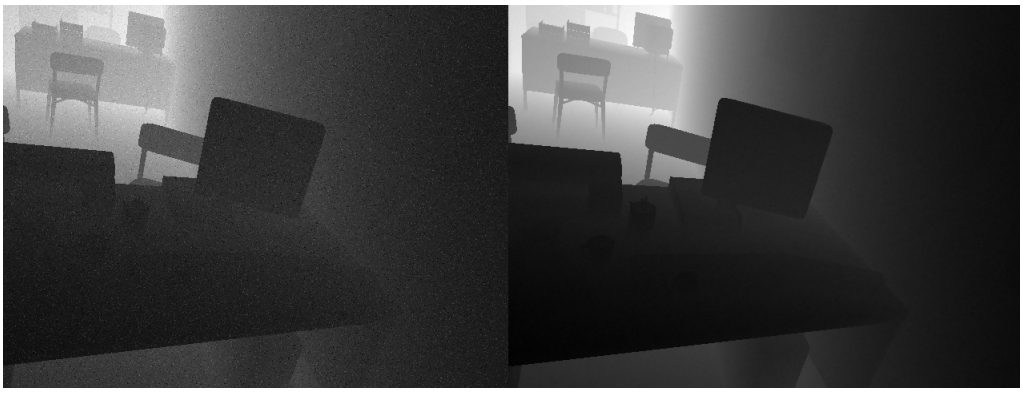


Figura 3.3: Antes y después de la fusión de la primera prueba. En cuanto al ruido $\sigma = 0{,}1$ y el 10 % de los datos son espurios con un variación máxima de 1 metro.

Partiendo de un mapa de profundidad cualquiera se le introducirá ruido gausiano con $\sigma = 0{,}1$ y datos espurios para el 10 % de los píxeles con una variación de 1 metro. Se realizará este proceso seis veces, tras lo que se realizará la fusión de todos ellos.

En la figura 3.3 se muestra uno de los mapas con ruido, ya que los otros cinco son visualemente similares, y el mapa resultado.

Ya a simple vista el resultado es muy bueno. En cuanto al análisis señal ruido, el resultado es también muy asombroso, ya que para los mapas ruidosos tiene un valor de entre 20 y 21 decibelios, pero para el mapa resultado devuelve un valor de 40 decibelios.

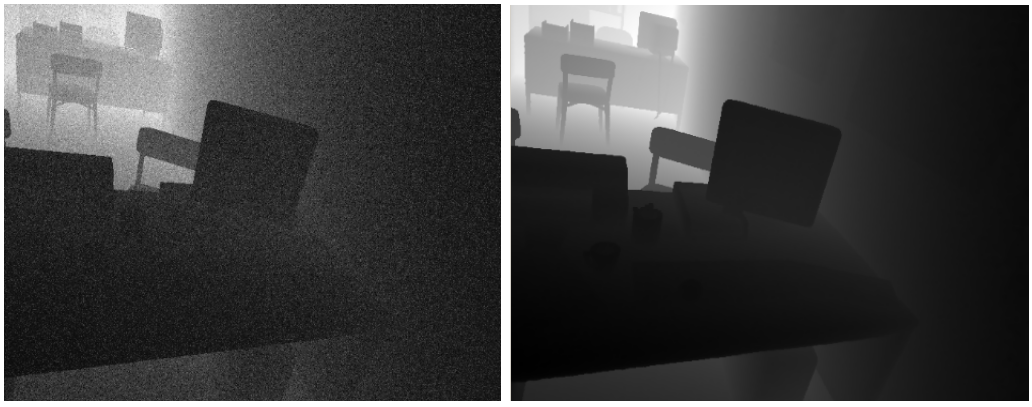
En la siguiente prueba ya entra en juego la virtualización. Se va a partir de un mapa original con un ruido gausiano de $\sigma = 0{,}1$ y datos espurios en el 50 % de los píxeles con variación de un metro. Para intentar mejorarlo se introducirán 11 mapas más, vistos desde distintas perspectivas con un ruido de $\sigma = 0{,}03$.

En la figura 3.4 se observan las relaciones señal ruido del mapa inicial ruidoso, y del resultado tras realizar la fusión.

Se puede observar una gran mejora tanto visual como en relación señal ruido. Cabe destacar que la prueba se ha realizado para una resolución de 512x512x512 del cubo. Si se aumenta esta resolución a 1024x1024x1024, el resultado mejora hasta una SNR de 25.55 dB.

En este punto se van a comenzar a realizar pruebas con mapas estimados. Estos mapas estimados se han obtenido mediante una técnica similar al $DTAM$ [RAND11] usando 10 imágenes en niveles de gris para cada depth map. En la figura 3.5 se muestra la diferencia entre un mapa perfecto y su correspondiente mapa estimado.

A continuación se va a probar que de la fusión de 12 mapas estimados resulta

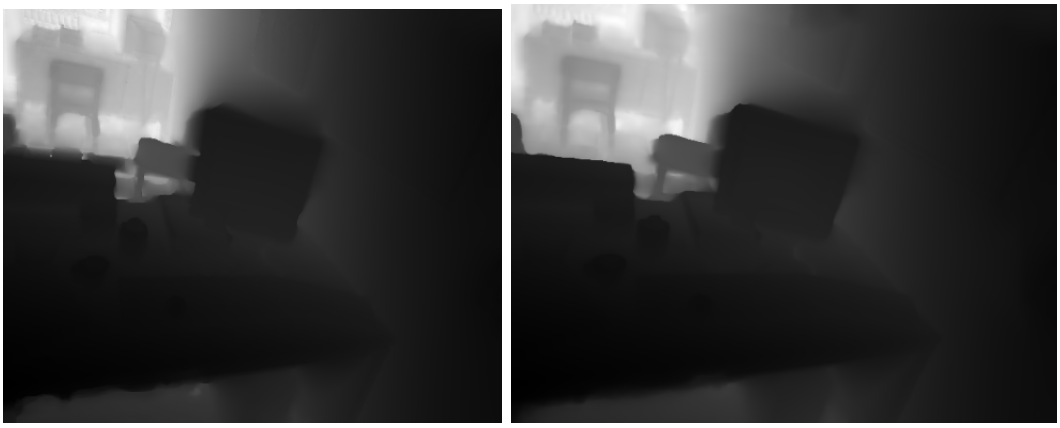


(a) Ruidosa con $\sigma = 0{,}1$ y espurios en el 10 % de los píxeles. snr= 15.16

(b) Resultado. snr=23.44

Figura 3.4: Resultado de la mejora de un mapa muy ruidoso mediante la fusión.

Figura 3.5: Comparación entre un depth map perfecto y su estimación correspondiente tras usar 10 imágenes consecutivas en niveles de gris



(a) Mapa estimado. snr= 17.5 (b) Resultado. snr=19.1

Figura 3.6: Resultado de la fusión de 12 mapas estimados.

uno más parecido al ground truth, por lo que la SNR debería aumentar en este caso. Los resultados de esta prueba se muestran en la figura 3.6.Se confirma el aumento de la SNR tras la fusión de los mapas.

Tanto en los bordes de la mesa como en los de la impresora se aprecia una mejora a simple vista. En la parte del monitor no corrige este efecto debido a que todos los mapas estimados tienen una mala estimación para esta parte, por lo que la solución sería mejorar el método de creación de depth maps.

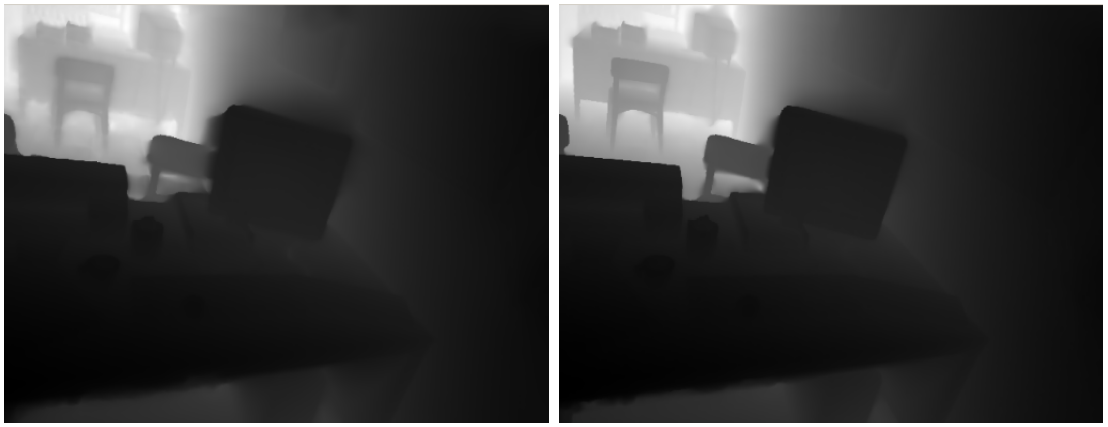
Para comprobar que la fusión puede resolver este problema se introducen mapas de profundidad de una buena calidad dentro de la fusión el resultado debería mejorar en gran medida.En la figura 3.7(a) se muestra el mapa obtenido tras fusionar un 85 % de mapas estimados y un 15 % de mapas de mas calidad y la SNR del resultado, y en la figura 3.7(b) el resultado y el SNR para un 50 % de mapas estimados y un 50 % de mapas de mas calidad.

La figura 3.8 muestra el efecto que tiene introducir mapas de mas calidad en la SNR. Como se esperaba se puede decir que la fusión mejora conforme la cantidad de mapas buenos introducidos aumenta. Otro efecto curioso es que la mejora depende del orden en el que se introducen debido al nivel de solapamiento respecto al depth map de referencia. Se observa con claridad que cuando se introduce el depth map numero 7 se produce una gran mejora como se observa en la pendiente de la gráfica.

3.4.3. Medida de tiempos

En cuanto a la medida de tiempos el interés se centra en dos procesos.

- Cuanto cuesta crear un solo mapa virtual, ya que que para una aplicación real este tiempo esta espaciado y depende del número de imágenes en niveles de



(a) Resultado. snr=19.53 (b) Mapa estimado. snr= 21.8

Figura 3.7: Resultado de la fusión habiendo añadido distinto porcentaje de mapas de mas calidad.

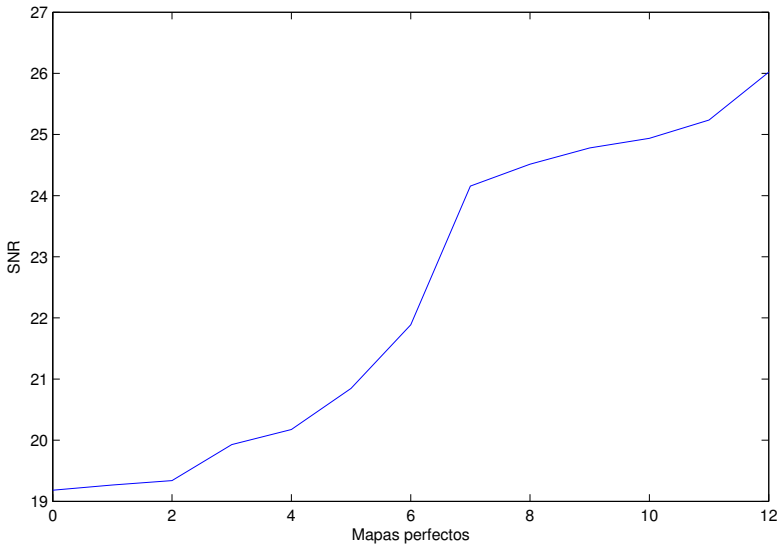


Figura 3.8: gráfica de mapas perfectos

gris necesarias para construir el depth map. Este tiempo es de una media de 200 milisegundos, ya que depende en cierto modo del mapa que se introduce en el cubo.

Cabe destacar que esta prueba se ha realizado para una resolución de 1024x1024x1024. Una bajada de esta resolución reduciría significativamente el tiempo de virtualización, por ejemplo para 512x512x512, el tiempo desciende hasta los 32 milisegundos (Tiempo real).

- Por otro lado es interesante saber el tiempo dedicado a la fusión en función de el número de mapas que se fusionan. En la figura 3.9 se observa este dato.

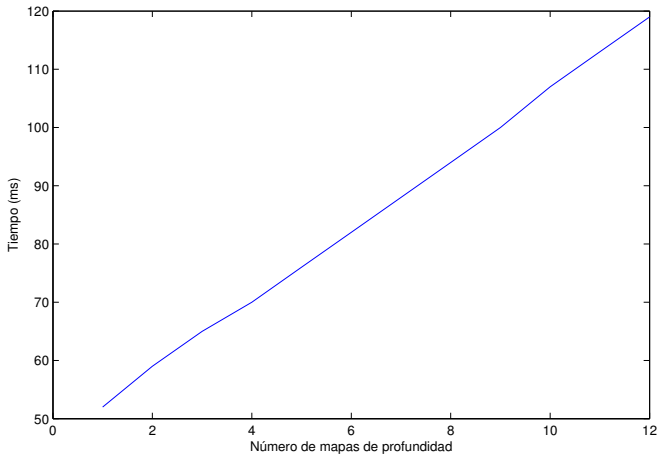


Figura 3.9: Tiempos de realización de la fusión en función del número de mapas implicados.

Esta prueba se ha realizado para 100 iteraciones un valor que asegura un buen resultado.

Estas pruebas, así como todas las de la memoria se han realizado sobre una tarjeta gráfica *nVidia Tesla M2090*.

Capítulo 4

Conclusiones

A lo largo de proyecto se han ido cumpliendo cada uno de los objetivos planteados al inicio del mismo. A continuación, se listan las aportaciones realizadas por parte de este PFC.

- Explicación sencilla de los cálculos matemáticos involucrados en todo el proceso, desde el algoritmo utilizado para la resolución del Primal Dual, hasta su aplicación en concreto para métodos de denoising y fusión.
- Estudio detallado de todos los parámetros implicados en el denioising, tanto de convergencia como de ponderación. De este modo se tiene una idea más clara para futuros trabajos sobre las elecciones a tomar en este aspecto, ya que la documentación existente no hace énfasis en este tema.
- Estudio de la posible aplicación en tiempo real de los algoritmos de denoising. Útil para aplicaciones posteriores.
- Aplicación del proceso de TGV-Fusion para imágenes proyectivas y no solamente ortográficas como se había realizado hasta la fecha.
- Estudio de los resultados de la fusión sobre la influencia del punto de vista y de distintos valores y tipos de ruido.

4.1. Lineas Futuras

En cuanto a posibles trabajos posteriores con relación de este proyecto:

- Estudio de la posible mejora del proceso de fusión mediante aplicación de distintos métodos de denosing después de la virtualización de los mapas, justo antes de la fusión.

- Realización de un promedio de los mapas directamente en el cubo que realiza la virtualización. En vez de transformar cada uno por separado. Comparar este resultado con el obtenido mediante la técnica utilizada en este proyecto.
- Realización de un método variacional con los datos directamente cargados en el cubo de virtualización, y no a posteriori como se realiza en este caso.
- Estudio de la distribución del trabajo enviado a la tarjeta gráfica, para optimizarlo según la arquitectura en la que se ejecuta.
- Por último, hubiera sido interesante la integración de la fusión realizada dentro de todo el proceso, desde la captura de imágenes hasta la reconstrucción de la escena. Y estudiar la posible ejecución en tiempo real de todo el proceso.

4.2. Valoración personal

Respecto a la valoración personal, este proyecto ha supuesto todo un reto, ya que se ha tratado del proyecto más grande que he realizado, gracias a esto he aprendido mucho, muchas veces de los propios errores.

También a supuesto un reto en cuanto a las herramientas empleadas, ya que antes de realizarlo carecía de experiencia en programación en Cuda. Tampoco nunca había realizado un documento en LaTeX lo cual ha hecho mucho más lenta la etapa de escritura de la memoria.

La realización de este proyecto me ha permitido conocer el mundo de la visión por computador, un área que a lo largo de la carrera no había tocado demasiado, pero que siempre me había llamado la atención. Además este proyecto se ha basado en la investigación, un campo en mi opinión muy interesante para cualquier ingeniero ya que es donde puedes aplicar todo tu ingenio, trabajar con tecnologías punteras, así como estar al día de todo, gracias a las distintas publicaciones de centros de investigación y universidades, en definitiva, el límite te lo pones tú mismo.

En cuanto al desarrollo del proyecto han sido aplicados conceptos de muy diversas materias aprendidas durante los últimos 5 años de carrera. Dentro de estos conceptos, por ejemplo los relacionados con la arquitectura de computadores, fueron muy útiles para comprender la arquitectura de la GPU, y de este modo programar de forma más eficiente. Por otro lado, se han aplicado multitud de conceptos relacionados con el cálculo y el cálculo numérico. En definitiva me he dado cuenta de que toda la formación recibida a lo largo de la carrera tiene una gran aplicación.

En cuanto a mi valoración personal propiamente dicha, ha sido una experiencia muy buena, ya que el tema del proyecto me gustaba, aunque en ocasiones lo vi

demasiado teórico, más tarde me di cuenta de que la teoría matemática es la base de todos los problemas relacionados con este tema, y que es necesaria una buena comprensión de la misma para poder abarcar cualquiera de los problemas resueltos a lo largo del proyecto.

Bibliografía

[AWB] Gottfried Graber Thomas Pock Andreas Wendel, Michael Maurer and Horst Bischof. Dense reconstruction on-the-fly.

[Bra00] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[BV04] Elsevier Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, Marzo 2004.

[CP11] A. Chambolle and T. Pock. A first-order primal-dual algorithm for convex problems with applications to imaging. *Journal of Mathematical Imaging and Vision*, 40(1), Mayo Mayo 2011.

[CUD10] *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 2010.

[LIRF92] S. Osher L. I. Rudin and E. Fatemi. Nonlinear total variation based noise removal algorithms. *Proc. of the 11th annual Int. Conf. of the Center for Nonlinear Studies on Experimental mathematics : computational issues in nonlinear sciences*, 1992.

[NY04] Nesterov and Yurii. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer, 2004.

[RAND11] S. J. Lovegrove R. A. Newcombe and A. J. Davison. Dtam: Dense tracking and mapping in real-time. *Proceedings of the 2011 International Conference on Computer Vision ICCV*, 2011.

[Roc97] R. T. Rockafellar. *Convex Analysis*. Princeton Math. Series, Princeton Univ. Press, 1997.

[SI11] Otmar Hilliges David Molyneaux Richard Newcombe Pushmeet Kohli1 Jamie Shotton1 Steve Hodges1 Dustin Freeman1 5 Andrew Davison2 Andrew Fitzgibbon Shahram Izadi, David Kim. Kinectfusion: Real-time 3d reconstruction and interaction using a moving depth camera.

Proceedings of the 24th annual ACM symposium on User interface software and technology., 2011.

[Sum10] Mark Summerfield. *Advanced Qt Programming.* Prentice Hall, 2010.

[TPB11] Lukas Zebedin Thomas Pock and Horst Bischof. Tgv-fusion. 2011.

Apéndice A

Optimización convexa: Algoritmo de Primal Dual

La optimización de funciones es una rama muy importante de estudio en las matemáticas, en general, no es posible encontrar el mínimo o el máximo absoluto de un problema de optimización aunque sí existen, para determinados tipos de funciones, soluciones óptimas o buenas heurísticas. [NY04]

Afortunadamente en este proyecto nos encontramos en uno de esos casos, en los que las funciones a minimizar que nos vamos a encontrar, presentan una estructura especial que permite obtener la solución global (óptima) del problema. La estructura genérica de los problemas que vamos a resolver, tanto en denoising como en TGV-fusion, es la siguiente:

$$\min_u F(Ku) + G(u) \tag{A.1}$$

Donde F y G son funciones convexas aunque no tienen por que ser diferenciables y K es un operador lineal. La solución a esta optimización se basa en el algoritmo Primal Dual que como veremos más adelante, cuando se aplica a problemas de visión por computador, se puede acelerar considerablemnte usando tarjetas gráficas (GPUs) y programación paralela en CUDA.

A.1. Algoritmo Primal Dual

Citando a [Roc97] "La gran línea divisoria en optimización no es entre problemas lineales y no lineales sino problemas convexos y no convexos". Para un número considerable de problemas convexos con cierta estructura es posible encontrar una solución óptima con buena precisión y en un tiempo razonable, independientemente de la inicialización. En nuestro caso la expresión 2.1 está compuesta por dos funciones convexas que admite una solución óptima [CP11].

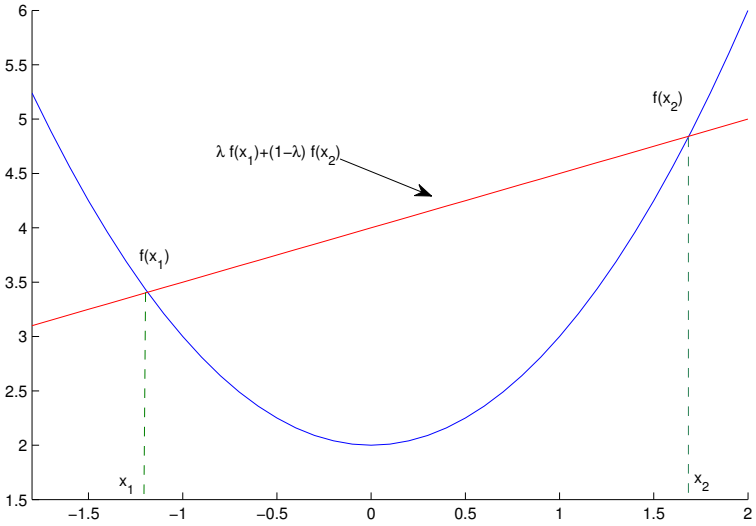


Figura A.1: Demostración de función convexa

Como vemos el concepto de convexidad es fundamental en el análisis y resolución de los problemas de optimización. Una función es convexa cuando se cumple que:

$$\forall \lambda \in [0,1] \text{ y } \forall x_1, x_2 \in dom(f)$$

$$\lambda f(x_1) + (1-\lambda) f(x_2) \geq f(\lambda x_1 + (1-\lambda) x_2) \tag{A.2}$$

Lo que viene a decir que dadas x_1 y x_2, el segmento de recta que une $f(x_1)$ y $f(x_2)$ siempre queda por encima que la función. En la figura A.1 se muestra gráficamente esta afirmación. La principal consecuencia de esto es que se puede encontrar en ella el mínimo absoluto, a continuación se muestran las técnicas que se van a emplear para conseguir esta minimización.

Una cualidad conveniente de toda función sobre la que se va a realizar una minimización es que sea diferenciable. En este punto es donde comienza el problema, ya que en la forma A.1 no se asegura que las funciones implicadas sean diferenciables. Sin ir mas lejos en el primero y más sencillo de los problemas de denoising que se abarcarán en el capítulo 3 esta función corresponde a la función valor absoluto. Para simplificar se va a suponer que la parte de $G(u)$ es diferenciable, así será para todos los problemas de denoising, pero no en el caso de la fusión de mapas que ya se verá en el capítulo correspondiente.

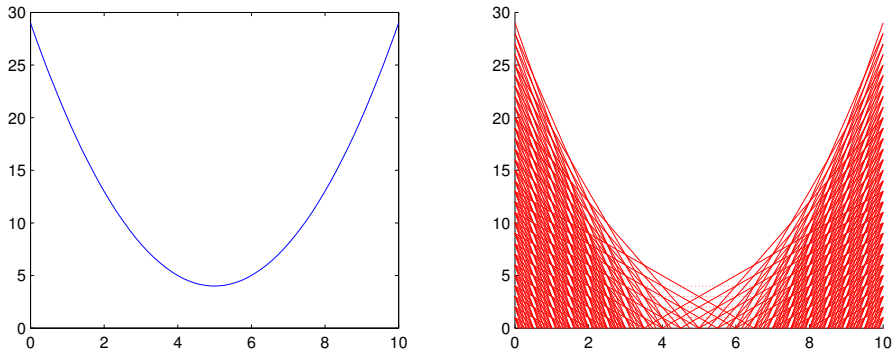


Figura A.2: Familia de tangentes que quedan por debajo de la función a estudiar

Para solucionar este problema se va a realizar una transformación de la función problemática de modo que sea compatible con los métodos de optimización clásicos.

A.2. Transformada de *Legendre-Fenchel*

Como se ha dicho no es posible aplicar los métodos clásicos a priori para resolver la minimización A.1, ya que no se asegura la diferenciabilidad de las funciones implicadas. Para lograr esto se va transformar la parte no diferenciable en una función apta para la aplicación de estos métodos para ello utilizaremos la transformación de *Legendre-Fenchel*.

En resumen, este método trata de definir las funciones convexas principalmente de otra forma sin perder información, en nuestro caso resulta especialmente útil, ya que va a permitir convertir una función convexa no diferenciable, en otra a la que es posible aplicarle el método del gradiente.

La transformación de *Legendre-Fenchel* trata de relacionar una función convexa con la familia de hiperplanos que quedan por debajo de ella es decir, es la intersección de todos los hiperplanos menores que la función a transformar.

En la figura A.2 podemos observar como la función queda también definida en este caso. El siguiente paso será tomar de todas estas intersecciones de las tangentes sólo las que coincidan con la propia función, por eso matemáticamente se expresa como el superior de todo lo anterior como veremos mas adelante, gráficamente queda bastante claro en la figura A.3 donde se muestran las tangentes a la propia función, que también definen a esta.

La forma vectorial de expresar cada uno de estos hiperplanos es $< x, p > -f(x)$, siendo $< x, p >$ el producto vectorial y, para este caso en particular, $< x, p >= x^T p$ ya que se trata de vectores. El resultado final de la transformación sería el siguiente.

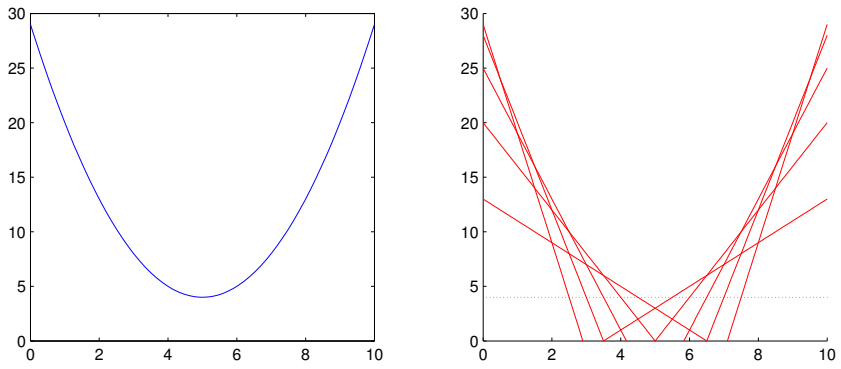


Figura A.3: Familia las rectas tangentes a la función a estudiar

$$f^*(p) \sup_x < x, p > -f(x) \tag{A.3}$$

A esta función $f^*(x)$ la llamaremos función dual de $f(x)$, es también de una función convexa. Ahora se aplica una propiedad de este tipo de funciones que dice que toda función convexa se puede poner en función de su dual. Tras esto resultará una expresión de la siguiente forma:

$$f(x) = \sup_p < x, p > -f^*(p) \tag{A.4}$$

Esta función es equivalente a la función inicial con la peculiaridad de que a esta le podremos aplicar los distintos métodos de minimización.

A.2.1. Valor absoluto

Como se ha comentado en el apartado de características, para uno de los algoritmos que se verán mas adelante la función $F(Ku)$ a la que se va a aplicar esta transformación es la función valor absoluto. A continuación se van a realizar todos los pasos para convertir la función valor absoluto en diferenciable. Esto es útil por un lado para ilustrar en un ejemplo lo explicado anteriormente, así como para partir de este punto al explicar en ocasiones posteriores el cálculo de la transformación dual de funciones parecidas al valor absoluto, útiles en distintos métodos de denoising.

En primer lugar se define la expresión general de la función dual que ha sido calculada en el apartado anterior.

$$F^*(p) = \sup_x < x, p > -F(x) \tag{A.5}$$

Para este caso en particular:

$$F^*(p) = \sup_x < x, p > -|x|_1$$

Al buscar el $\sup_x < x, p > -|x|_1$ estamos buscando el superior de $< x, p >$, es decir de $|x||p| \cos\theta$ siendo θ el ángulo que forman. Para obtener el superior de eso tendremos que tomar $\theta = 0$, por lo que la expresión A.5 se reduce a:

$$F^*(p) = \sup_x |x||p| - |x|$$

O lo que es lo mismo:

$$F^*(p) = \sup_x |x|(|p| - 1)$$

Esta última aproximación ya se puede interpretar como una función por partes. Para ello se puede observar el termino $(|p| - 1)$.

- Cuando $(|p| - 1)$ sea menor o igual a 0, el máximo posible para toda la expresión sera 0, esto se dará al tomar $x = 0$, ya que cualquier otra elección daría un resultado mas bajo.
- Si $(|p| - 1) > 0$, no existirá una cota superior ya que el máximo valor de la expresión será dado en el peor de los casos por el valor de $|x|$, por lo que el resultado para estos tramos es ∞.

En resumen, resulta de forma simbólica la expresión A.6, lo que se observa gráficamente en la figura A.4

$$F^*(p) = \begin{cases} 0 & |p| \leq 1 \\ \infty & |p| > 1 \end{cases} \tag{A.6}$$

Por último, se pone la función inicial $f(x)$ en función de su dual, con lo que resulta:

$$f(x) = \sup_p < x, p > -F^*(p)) = sup_{|p|\leq 1} < x, p >$$

Dando como resultado una maximización de una función diferenciable con una restricción $(|p| \leq 1)$ con la que se puede trabajar, más adelante se explicará el por qué de esta función y se describirá completamente.

Para concluir se va a realizar una análisis gráfico de la solución alcanzada. En la figura A.5 se observa por un lado, en azul, la función valor absoluto, por otro lado se muestran dos funciones tangentes a la misma, cada una es un ejemplo cada una de las dos partes en las que se divide la función dual de la función original.

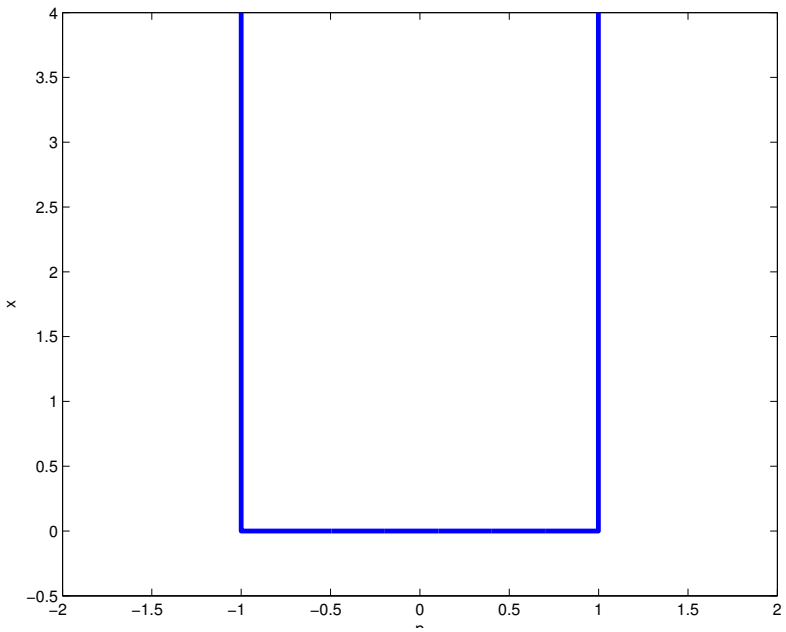


Figura A.4: Dual de la función valor absoluto

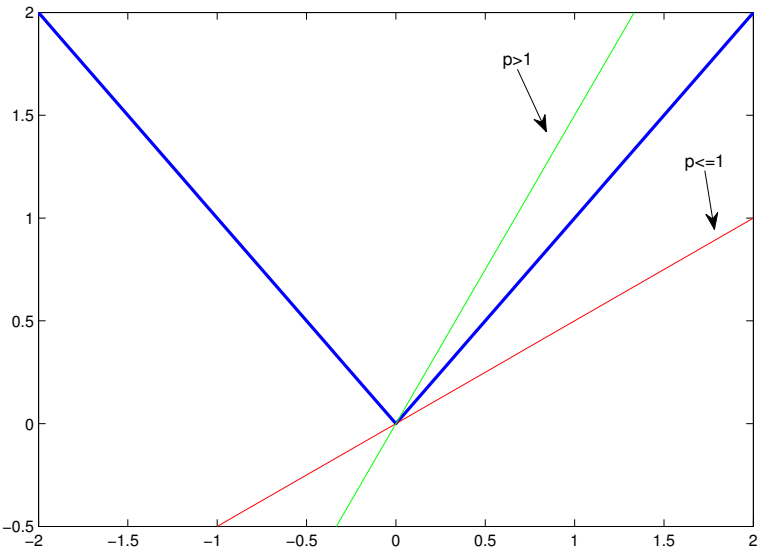


Figura A.5: Ejemplo útil para comprender la dual de la función valor absoluto

Para comprender mejor la figura hay que recordar la ecuación A.3, en la que hay dos términos diferenciados, que podemos distinguir en la figura. En primer lugar está el término $< x, p >$ representado por las rectas que pasas por el origen representadas con los colores rojo y verde, y por otro lado está la propia función valor absoluto representada en azul. Lo que se pretende buscar es el máximo de la resta de ambas partes.

- Para $p > 1$ vemos que para x positivas la recta queda por encima de la función, por lo que el resultado será positivo y mayor cuanto mas se aleja del origen, por lo que la máxima diferencia entre ambas partes será infinito.
- Por otro lado para $p \leq 1$, la tangente siempre queda por debajo de la función por lo que siempre será negativa salvo en el punto en el que es tangente, es decir, en x=0 cuya diferencia será 0.

A.2.2. Norma de Huber

Se puede definir la función de la norma de Huber como una función por partes de la siguiente forma.

$$F(x) = \begin{cases} \frac{|x|^2}{2\alpha} & |x| \leq \alpha \\ |x| - \frac{\alpha}{2} & |x| > \alpha \end{cases} \tag{A.7}$$

En este punto como se ha hecho con la función valor absoluto para el TV, se tiene que hallar el dual de la función A.7. En este caso la demostración matemática no es tan evidente como en el caso del valor absoluto por lo que se va a realizar una demostración gráfica basada en la figura A.6. En la figura se puede observar, por un lado la función A.7 en color azul y naranja, y por otro lado dos ejemplos de rectas tangentes a la función, como en el caso del valor absoluto una con $p > 1$ y otra con $p \leq 1$.

- Para el caso de $p > 1$ (recta verde) su superior es ∞, como se explicó en para el valor absoluto la diferencia máxima entre la función y la recta no está acotada.
- La novedad está en que para $p \leq 1$, ya que en el caso anterior la máxima diferencia era 0 porque siempre estaba por debajo de la función, pero en este caso hay una parte de la tangente que está por encima de la función, además se puede demostrar que la máxima diferencia existente siempre es entre la tangente y el tramo $[-\alpha, \alpha]$ y esta diferencia es $\frac{\alpha p^2}{2}$.

Por todo lo anterior la expresión resultante para el dual de la función A.7 es la siguiente:

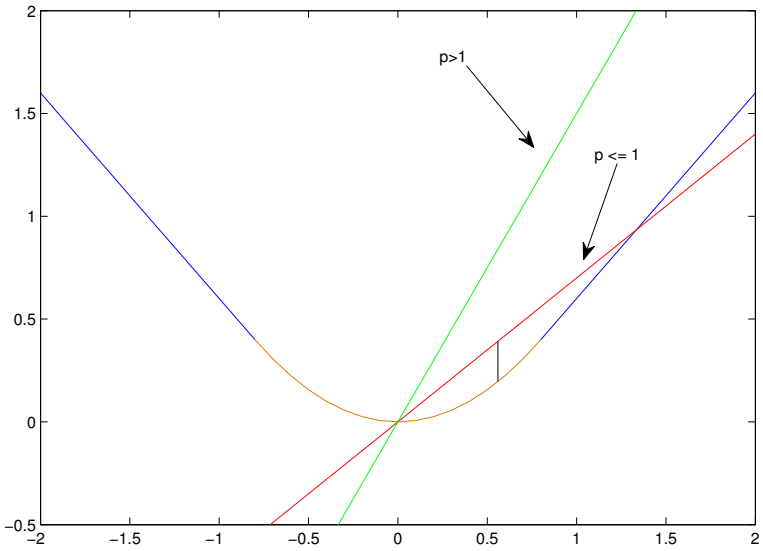


Figura A.6: Gráfica útil para comprender el dual de la función regularizadora de Huber

$$F^*(p) = \begin{cases} \frac{\alpha p^2}{2} & |p| \leq 1 \\ \infty & |p| > 1 \end{cases} \tag{A.8}$$

A.2.3. TGV

En el caso del TGV como en los métodos anteriores existe un cambio en la parte del regularizador, pero no se trata de un cambio de función, en este caso introduce una función nueva v que lo que pretende es dar cabida a las variaciones lineales del gradiente. Esto es, en superficies planas de la imagen en las que la variación de color sea lineal no habrá penalizaciones que le obliguen a generar pequeñas zonas de igual color como en el TV, si no que permitirá esta variación gradual del color, en la figura A.7, se observa este hecho en funciones sencillas.

Esta nueva variable v representa la variación del gradiente a lo largo de la imagen, por lo que como se ha dicho antes, minimizar esta función tiende a generar áreas de gradiente constate.

En la figura se observan cuatro gráficas de funciones significativas para entender el problema, la primera gráfica es la supuesta función de entrada con ruido, la segunda la solución que generaría el método TV, como se ha dicho antes se observa que tiende a generar superficies planas, en las que no varia el gradiente. En la

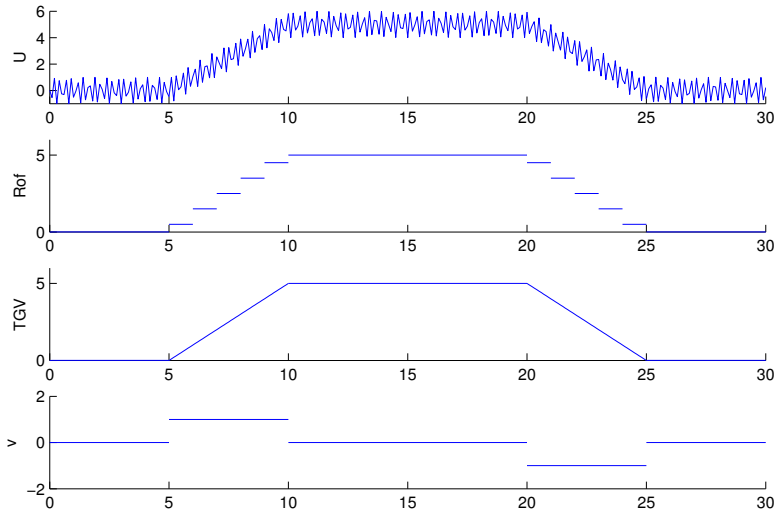


Figura A.7: Comparación de los distintos métodos en una función sencilla

tercera gráfica se observa el resultado que resultaría del método TGV, donde se conservan las pendientes como debería. Por último se representa la función v, que es la cual permite que el TGV se comporte de esa forma, porque el método minimiza la variación total de esta función como lo hacia TV para la propia imagen, por lo que generará en ella sectores planos que se corresponderán variaciones constantes del gradiente en la imagen resultado.

La parte del regularizador queda pues de la siguiente forma.

$$|\nabla u|_{TGV} = \alpha_1 |\nabla u - v| + \alpha_2 |\nabla v| \tag{A.9}$$

Al haber ahora dos términos existirán dos funciones a dualizar, la ventaja es que ambas son la función valor absoluto, por lo que esas funciones son sencillas como se ha visto en el caso del TV. A continuación se muestran ambas, como en casos anteriores p es el gradiente de u, y en este caso en particular q va a ser el gradiente de v.

$$F^*(p) = \begin{cases} 0 & |p| \leq \alpha_1 \\ \infty & |p| > \alpha_1 \end{cases}$$

$$F^*(q) = \begin{cases} 0 & |q| \leq \alpha_2 \\ \infty & |q| > \alpha_2 \end{cases}$$

A.3. Resolución del problema

A partir del apartado anterior, podemos concluir que la ecuación A.1 de la que se partía, tras aplicar la transformación de *Lengendre-Fenchel* para el término $F(Ku)$ resulta la siguiente expresión.

$$\min_n \sup_p < Ku, p > +G(u) - F^*(p) \tag{A.10}$$

$$\Downarrow$$

$$\min_n \sup_p < Ku, p > +G(u) \text{ sujeto a } |p| \leq 1$$

Ahora la expresión ya es diferenciable, ya que la parte $F(Ku)$ ha quedado transformada, y la parte $G(u)$ era diferenciable de inicio como se ha supuesto en el apartado de características. Por lo que resulta un problema de minimización, maximización de funciones diferenciables.

En este punto es posible aplicar métodos de primer orden (basados en que la función es diferenciable al menos una vez) para la optimización.

A.3.1. Optimización convexa de primer orden

En primer lugar se va a definir la optimización mediante métodos de primer orden ya que es el algoritmo en que se basará la solución del problema Primal-Dual para los problemas que se desean resolver. Interesa resolver el siguiente problema:

Dada $f : \mathbb{R}^n \longrightarrow \mathbb{R}$, campo escalar una vez diferenciable con continuidad, encontrar $\bar{x} \in \mathbb{R}^n$ solución de:

$$\min_{x \in \mathbb{R}^n} f(x) \tag{A.11}$$

Encontrar una solución de este problema corresponde a encontrar un punto $\bar{x}$ que satisfaga:

$$f(\bar{x}^*) \leq f(x) \forall x \in \mathbb{R}^n \tag{A.12}$$

El punto $\bar{x}^*$ se denomina mínimo absoluto de f sobre $\mathbb{R}^n$

Sabiendo que la función a estudiar es convexa y diferenciable, si pensamos en funciones de una variable, habremos encontrado un mínimo cuando $f'(x) = 0$, pero al extrapolar este problema a $\mathbb{R}^n$, podemos demostrar de forma teórica que nos encontramos en un mínimo cuando se cumple la siguiente condición necesaria y suficiente. [BV04]

- $\bar{x}$ es un punto estacionario de f

$$\nabla f(\bar{x}) = \vec{0} \Longleftrightarrow \begin{bmatrix} \frac{\partial f(\bar{x})}{\partial x_1} \\ \frac{\partial f(\bar{x})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\bar{x})}{\partial x_n} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

A continuación se va a describir el método del gradiente descendente que será el que se utilizará con ciertos matices para resolver el problema.

A.3.2. Método del Gradiente

El método del gradiente descendente es uno de los métodos sencillo de optimización de funciones diferenciables.

Se trata de un método iterativo que dado un punto inicial busca la dirección de máximo descenso y elije un paso para acercarse a la solución. Matemáticamente lo se puede ver como que en cada iteración se busca una dirección $d \in \mathbb{R}^n$ dada por la siguiente expresión.

$$\min_{d \in \mathbb{R}^n} f(x)d \text{ sujeto a } \parallel d \parallel^2 = 1$$

Con lo que resulta la siguiente expresión.

$$d = -\nabla f(x)$$

Es decir, la dirección local de máximo descenso es la dirección del gradiente negativo.

En resumen el método del gradiente descendente obedece a la siguiente expresión donde en cada paso hay que calcular tau usando un algoritmo de line search.

$$x^{n+1} = x^n - \tau \nabla f(x^n) \tag{A.13}$$

A.3.3. Otros métodos de optimización

Como se ha dicho anteriormente el método del gradiente es el método más sencillo, y como tal en condiciones normales es de los más lentos en alcanzar la solución, existen muchos otros métodos a priori más rápidos y eficientes. Por ejemplo uno de los más conocidos es el método de *Newton*, que se basa en el mismo concepto, pero realiza una aproximación cuadrática a la hora de elegir la dirección de descenso.

La clave para elegir el de gradiente y no el de *newton* es que en la práctica dado que cada componente del gradiente se puede calcular de forma independiente, se puede paralelizar totalmente, dada la independencia de los datos. Si se piensa en problemas relacionados con la imagen, para hilo de ejecución hará las operaciones para cada pixel, mientras que en el caso de métodos de aproximación cuadrática los datos no son independientes y resultan operaciones más complejas que involucran a más datos, por lo que no aprovecharíamos la potencia de la programación gráfica.

A.4. Proximal map

A pesar de que es posible la utilización del método del gradiente para la optimización de la expresión A.10, se va a utilizar un método llamado *proximal map*. Se trata de un método muy similar al del gradiente descendente pero incluye varias características que lo hace perfecto para la resolución del Primal Dual. Como se ha dicho en anteriores ocasiones se parte de la siguiente expresión general.

$$\min_x \sup_y < Kx, y > +G(x) \text{ sujeto a } |y| \leq 1$$

$$\Downarrow$$

$$\min_n \sup_p C(x, y) \text{ sujeto a } |y| \leq 1$$

Se muestra en función de una función de energía standard para facilitar la comprensión.

A continuación se muestra la estructura de este método.

$$\begin{cases} y^{n+1} = (I + \sigma\partial F^*)^{-1}(y^n + \sigma K\bar{x}^n) & (1) \\ x^{n+1} = (I + \tau\partial G)^{-1}(x^n - \tau K^* y^{n+1}) & (2) \\ \bar{x}^{n+1} = x^{n+1} + \theta(x^{n+1} - x^n) & (3) \end{cases} \tag{A.14}$$

Este es el esquema de la resolución del problema Primal Dual, en el se realiza una maximización y una minización para cada una de las iteraciones.

El la parte 1 del proceso se realiza la maximización. Como se hacía para el método del gradiente descendente se deriva en función de y.

$$\tilde{y}^{n+1} = y^n + \sigma\nabla_y C(x^n, y^{n+1})$$

En este punto se observa la primera de las ventajas del *proximal map*, evalúa a y en el punto siguiente del que se encuentra en la iteración actual, lo que acelerará el proceso.

La siguiente ventaja viene dada por el operador $(I+\sigma\partial F^*)^{-1}$, su labor consiste en realizar una proyección de modo que cumpla la restricción que impone la función dual, de lo que resulta lo siguiente.

$$y^{n+1} = \frac{\tilde{y}^n}{\text{máx}(1, \tilde{y}^{n+1})}$$

La parte 2 del proceso realiza la minimización en x, este caso en algo mas sencillo que el anterior ya que al realizar la primera derivada desaparece la función dual así como su restricción por lo que el operador $(I+\tau\partial G)^{-1}$ es trivial. En este caso sigue apareciendo la ventaja de que la fución es evaluada en el punto siguiente al realizar los cálculos por lo que la derivada tendrá la siguiente forma.

$$x^{n+1} = x^n - \tau\nabla_x C(x^{n+1}, y^{n+1})$$

En este caso y también se evalúa en el punto siguiente ya que puesto que este resultado es conocido de antemano.

En cuanto al tercer elemento no tiene una importancia vital para el problema, se trata de una relajación para lograr que el método converja con mayor rapidez.

En cuanto al paso, para esta forma de función de energía en concreto se puede demostrar que tomando τ y σ de forma adecuada se asegura la convergencia. Por lo que no será necesario realizar el cálculo del paso en cada iteración como si que lo sería con el método del gradiente.

En resumen el método *proximal map* utilizado se basa en el método del gradiente descendente y ascendente evaluados en el punto siguiente y con un paso conocido de antemano haciendo que en realidad la convergencia sea mas rápida y los cálculos más sencillos que en un método de gradiente tradicional.

A.4.1. Elección de parámetros

Una particularidad de la función de energía de este proyecto es que se puede asegurar que con una correcta elección de los parámetros el método de gradiente para este problema siempre converge. Esta correcta elección se refiere a que los distintos pasos τ y σ deben cumplir la siguiente expresión.

$$\tau\sigma L^2 \leq 1 \tag{A.15}$$

Siendo $L = \parallel K \parallel$ es decir, la norma de K. En el artículo en el que se basa esta parte del proyecto [CP11], escoge ambas con valor aproximado de 0,3536 ya que se puede demostrar que $\parallel K \parallel = \sqrt{8}$, por lo que haciendo en la expresión anterior $\tau = \sigma$ el resultado es el comentado anteriormente.

Apéndice B

Gestión del proyecto

En este capítulo del anexo se va a detallar todo el proceso que se ha ido realizando a lo largo del proyecto.

B.1. Metodología

Dadas las características del proyecto no se aplicó ninguna metodología concreta para su realización. A pesar de esto el desarrollo del proyecto se puede dividir en distintas fases.

Aprendizaje

En principio se realizó una etapa de aprendizaje, por un lado del algoritmo Primal Dual, y por otro del primero de los problemas de denoising, ya que se trata de el más sencillo, pero en el que se aplican todos los conceptos que se necesitarían para los siguientes problemas.

A la vez que se aprendían los conceptos teóricos y matemáticos, se comenzó a programar los distintos problemas en distintas tecnologías gradualmente hasta llegar a la el último escalón, una vez allí, se fueron implementando los demás métodos de denoising así como la fusión.

Se detallan a continuación todos los pasos llevados a cabo.

Matlab

Como se ha dicho antes se tomó el más sencillo de los problemas de denoising, el $TV\text{-}Rof$, para ello se estudió el problema propuesto en el Pock [CP11], y se programó de forma literal en primer lugar en lenguaje *matlab*, ya que mediante su uso las operaciones complejas con matrices se tornan muy sencillas. Por lo que no se tuvo en este primer acercamiento una visión a nivel de pixel, y por tanto era

imposible una interpretación paralela del método. Aun así esta etapa como se ha dicho ayudó a la compresión del problema y del algoritmo.

C++

Una vez que el problema estaba programado y funcionando en *matlab*, se comenzó a implementar en $C++$.

La principal novedad, y punto importante de $C++$ frente a *matlab*, hubo que cambiar la concepción del problema, había que pensar a nivel de pixel.

El paso de pensar en matrices a pensar en píxeles, fue lo mas complejo de esta etapa pero a la vez lo más interesante ya que se veía claramente que se podía paralelizar de forma sencilla.

CUDA

El siguiente paso, y último, fue la implementación en *CUDA*. La principal ventaja del uso de GPU's es, como se comenta en la memoria, la gran paralelización que se logra, ya que las funciones primal y dual se ejecutan una vez por cada pixel, y en cada uno de estos kernels[1] se realizan operaciones sencillas en las que las GPU's marcan la gran diferencia.

Una vez se llegados a este punto, se fueron realizando los siguientes algoritmos, después de tenerlos todos implementados, se inició la realización de las aplicaciones que les daban soporte a estos métodos.

Más adelante en los anexos correspondientes se detalla el proceso seguido para la implementación de cada una de las interfaces.

B.2. Tecnologías empleadas

La elección de las tecnologías empleadas se basó principalmente en las directrices dadas por el director del proyecto, ya que todos los trabajos en ese grupo de investigación se basaban en las mismas tecnologías por lo que tenían experiencia en posibles problemas que pudieran surgir.

La principal tecnología empleada, y que marcó la diferencia es $Cuda/C++$ utilizada para el backend de la aplicación, es decir, la programación de todos los problemas propuestos en los distintos Papers.

Como se ha comentado en otras ocasiones a lo largo de la memoria, gracias a la independencia de las operaciones para cada pixel en el cálculo del Primal Dual, se logra una aceleración, y una gran mejora en el tiempo de ejecución respecto a versiones en $C++$ o *matlab*, por ejemplo para *matlab*, solo dos iteraciones, le

[1]kernel: Función que se ejecuta en la gpu de forma paralela

costaba casi treinta minutos aunque la mayor parte de este tiempo estaba creando una matriz para realizar el cálculo del gradiente. En el caso de $C++$ sin paralelizar para unas 200 iteraciones tardaba sobre 4 segundos. Por último en la versión más rápida de *cuda* llega a hacer las 200 iteraciones en treinta milisegundos, es decir, en tiempo real.

A continuación se explica de forma resumida que es *Cuda*, y que características tienen la arquitectura Nvidia.

Cuda es una plataforma diseñada conjuntamente a nivel software y hardware para aprovechar la potencia de una GPU en aplicaciones de proposito general, permite una comunicación sencilla entre CPU y GPU, y su hardware esta orientado al cálculo paralelo.

El paradigma de programación de cuda consiste en programar ciertas funciones a paralelizar tal y como se haría si se quisiese ejecutar en un solo hilo, y el sistema se encarga automáticamente de ejecutar esta función paralelamente en tantos procesadores de la GPU como se requiera.

En la figura B.1 se puede observar una visión global de la arquitectura $Fermi$, la tercera generación del hardware paralelo de $Nvidia$, en ella se distinguen a primera vista una serie de multiprocesadores, y los distintos niveles de memoria.

Cada uno de los multiprocesadores esta compuesto en este caso de 32 núcleos que podrán ejecutarse en paralelo, así como memoria caché local y distintas herramientas para planificar la ejecución de los hilos.

En cuanto a la programación en *Cuda* como se ha comentado antes, es muy parecida a la programación secuencial en CPU, la novedad está en el compilador *nvcc* que es el que selecciona las partes que han sido programadas para la *gpu* (*Funciones kernel*) y las compila para su ejecución ella. Las interacciones de datos entre la CPU y la GPU se realizan mediante un bus PCI.

Otro concepto importante para la programación en cuda es el uso de distintos tipos de memoria, que serán útiles para acelerar el proceso. En la figura B.2 se observa la jerarquía entre los distintos niveles de memoria. Para comprender esto hay que tener claro la forma que *cuda* divide el trabajo para enviarlo a los distintos núcleos de la GPU. *Cuda* estructura el trabajo en varios niveles.

- El hilo o thread es el nivel más bajo, cada uno de ellos se ejecutará en un núcleo.
- Estos *threads* se agrupan en bloques generalmente de 32. Cada uno de estos bloques se ejecutará exclusivamente en un multiprocesador.
- Por último el nivel más alto se conoce como grid, será quien agrupe todos los bloques de una ejecución.

De todas las memorias de la figura se usarán a lo largo del proyecto las que son visibles para todos los multiprocesadores que trabajan con el mismo grid, es decir:

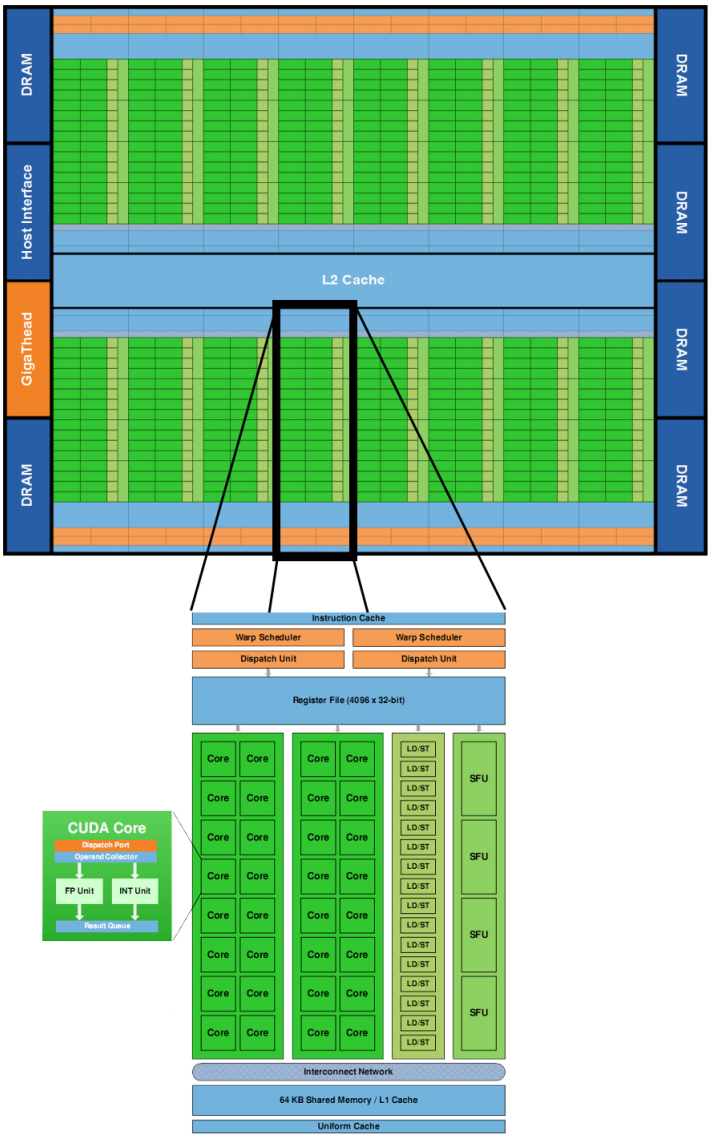


Figura B.1: Visión global de la arquitectura *Fermi* de *Nvidia*

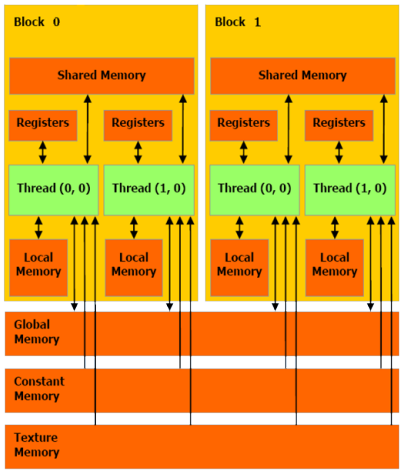


Figura B.2: Jerarquía de memoria para la programación en *cuda*

- Memoria global: Es la única memoria de lectura y escritura visible en todo el *grid*.
- Memoria constante: Es una memoria exclusivamente de lectura, de poco tamaño, útil para alojar en ella parámetros que se mantendrán constantes a lo largo de la ejecución y serán consultados con asiduidad por la *GPU*.
- Memoria de textura: Es, como la memoria constante, de solo lectura, es más lenta que esta, pero tiene mucha más capacidad. Además, esta optimizada la localidad 2D, lo que hará que sea idónea para la consulta de píxeles de una imagen cercanos a el que se esta trabajando.

Para el frontend de las distintas aplicaciones de soporte a los problemas expuestos en los papers se utilizó QT [Sum10], ya que se trata de librería multiplataforma ampliamente usada para desarrollar aplicaciones con interfaz gráfica de usuario, por lo que existe una gran comunidad que ayudan a solucionar todos los posibles problemas, ademas de eso es lo que se ha usado siempre en el grupo de investigación, por lo que fue una elección fácil.

La principal diferencia entre *QT* y otros frameworks de interfaces, es que no trabaja con eventos y funciones que capturan esos eventos. Tiene un sistema de señales y slots, que básicamente es lo mismo pero dicho de otro modo. Antes de iniciar el programa se han de realizar las conexiones, esto es indicar que slot, o slots, se ejecutarán una vez se haya emitido cada señal.

Existen principalmente señales de dos tipos, por un lado están las que produce la interfaz, que en funcionamiento serian similares a los eventos de otro tipo de sistemas, por ejemplo, cuando un botón es clickado, cuando una caja de texto ha

sido modificada, etc. Por otro lado están las señales programadas por el usuario, que son las que marcan la diferencia, ya que permiten por ejemplo interacción entre clases, e incluso son un posible canal de comunicación al permitir enviar parámetros que recibe la función slot que captura la señal.

A pesar de que C++, Cuda y Qt, fueron las principales tecnologías empleadas, no son las únicas, cabe destacar también el uso de matlab en los inicios del proyecto como se ha comentado en el apartado anterior, ya que se trata de un lenguaje muy potente para temas matemáticos principalmente en operaciones con matrices, útiles para este tipo de problemas. Mas allá de la potencia del lenguaje de programación, matlab ha sido usado para realizar las gráficas explicativas de la memoria ya que también es muy potente y flexible en esa faceta.

B.3. Herramientas utilizadas

En cuanto a las herramientas utilizadas a lo largo del proyecto son principalmente las que dan soporte a las tecnologías detalladas en el apartado anterior.

Generalmente a la hora de programar se ha utilizado el *QtCreator*, ya que por un lado sirve como IDE de $C++$, y por otro incluye el *QtDesigner* que se trata de una herramienta muy útil, por no decir indispensable para construir la interfaz de forma sencilla.

En cuanto al uso de *Cuda*, a pesar de que no había problema a la hora de usarlo con el *QtCreator*, en un principio no se disponía de un ordenador con tarjeta gráfica compatible, lo que complico el trabajo sobremanera.

Para subsanar este problema había que conectarse a un cluster del i3a donde existían nodos que disponían de tarjetas gráficas. Las tarjetas gráficas eran del modelo $Tesla\ M2090$.

El hecho de tener que programar en una máquina, y compilar en otra dió multitud de problemas, el primero fue que las librerías que se utilizaban en local, principalmente la de *OpenCV* [Bra00], tenia una versión antigua, con lo que hubo que subir las librerías compiladas y los include a mano.

Como herramienta para la compilación se utilizó CMake. CMake es un sistema de generación de scripts de compilación (build system) que utiliza ficheros de configuración independientes de plataforma y de compilador para generar el proyecto. Gracias a esto, con el mismo CMake se puede compilar en local, y compilar y ejecutar en el servidor.

Todo lo anterior propició el uso de mas herramientas como fueron, *Filezilla* para el intercambio de archivos, *Terminal* para ejecutar y manejar el servidor mediante *ssh*, Coda 2 para el intercambio de ficheros y la edición de los mismos mediante $MAC\ OSX$ entre otras.

Este problema se agudizó en cuando cerraron el cluster por cierre energético

durante gran parte de agosto, cuando todavía no se había finalizado la programación, por lo que hubo que conseguir un antiguo ordenador que disponía de una tarjeta gráfica compatible (*GeForce* 9600 *gt*), que era sobre unas 10 veces más lenta que las tarjetas alojadas en el cluster, pero válida para compilar y ejecutar los programas.

Para la escritura y edición de la memoria se ha utilizado Texmaker, se trata de un editor LaTeX muy potente y de libre distribución. Como se ha dicho antes los gráficos están hechos con *Matlab*, y algunas de las figuras con *Adobe Photoshop*.

Apéndice C

Programación de Primal Dual

Una gran parte del proyecto se centró en la programación de los distintos problemas de denoising y fusión, esta fue la parte más elaborada de toda la programación y a su vez la mas delicada y difícil.

Existen un par de factores que hicieron que esta parte fuera una de las que mças horas requiriera.

- Por un lado, es una tecnología totalmente nueva para mi, por lo que hubo una gran parte del tiempo dedicada al aprendizaje, para ello fue muy interesante la lectura del libro de referencia [CUD10], en el que define perfectamente todos los elementos de la tecnología y además esta repleto de ejemplos muy útiles para comenzar a programar.

- El otro gran problema que surgió fue la dificultad para encontrar errores en las funciones *kernel*, ya que para consultar el resultado de cualquier operación hay que esperar al fin de ejecución y descargar el resultado.

  Además al tratarse de problemas en imágenes la mayoría de las veces ver un mal resultado no da pistas de lo que haya podido pasar.

Se va a comenzar explicando paso a paso la estructura del código del algoritmo de denoising TV ya que como se ha dicho con anterioridad es el más sencillo de todos. Además los cambios que hay que realizar sobre este para programar los demás son mínimos en cuanto a la estructura del algoritmo, estos cambios están localizados principalmente en las funciones *kernel* que se explicarán más adelante.

```
1 Mat TV(Mat Inoisy, float tau, float sigma, float lambda, int
    iteraciones)
2 {
3     //Declaracion de los punteros con los que trabajara la cpu
4
```

```
5       //Declaracion de los parametros del metodo
6       float gamma = 0.7f*lambda;
7       float theta;
8
9       //Reserva espacio en gpu para los datos necesarios
10
11      cudaMalloc(...);
12      ...
13
14      //Inicializacion de los datos
15
16      cudaMemcpy( ... , cudaMemcpyHostToDevice ) ;
17      cudaMemset(...);
18
19    //Carga de las texturas
20      cudaBindTexture2D(...);
21      ...
22
23    //Parametros de las dimensiones de trabajo
24      dim3 nThreads(NTHREADS_BLOCK, NTHREADS_BLOCK);
25      dim3 nBlocks(divUp(Inoisy.cols,nThreads.x), divUp(Inoisy.rows,
        nThreads.y));
26
27      //Bucle de ejecucion
28      for (int iter = 0; iter < iteraciones; ++iter){
29
30          updateDualGPU <<< nBlocks , nThreads >>> (...);
31
32          theta = 1.0f/sqrtf(1.0f+2*gamma*tau);
33
34          updatePrimalGPU <<< nBlocks , nThreads >>>(...);
35
36          tau = theta*tau;
37          sigma = sigma/theta;
38       }
39
40       cudaMemcpy(Iresult.data, gpu_u, nbytes, cudaMemcpyDeviceToHost);
41
42      //Liberacion de las memorias de textura
43      cudaUnbindTexture(...);
44
45      //Liberacion la memoria en la gpu
46      cudaFree(..);
47
48      return Iresult;
49  }
```

El código anterior es bastante autoexplicativo, en él principalmente hay tres zonas.

- La primera de declaración de variables ya carga de datos, lo único a destacar en esta parte son las instrucciones finales donde inicializa los parámetros para la llamada al *kernel*.

  En resumen lo que hace en esta parte es asignar en primer lugar el número de threads que serán generados para cada bloque, para ello declara una malla de 16*16 threads, después calcula el número de bloques que serán necesarios para recorrer toda la imagen.

  Más tarde en la llamada a la función kernel se le indica estos dos valores, para que se realice la paralelización deseada.

- En la parte central esta el bucle de minimización. En cuanto a este bucle solo ejecuta la actualización del primal y del dual una vez por cada iteración, indicando los bloques y *threads* necesarios para una máxima paralelización como se acaba de explicar.

  La estructura solo cambia ligeramente en el caso de método *Huber*, ya que como se ha dicho con anterioridad este método admite una variación que hace que sea mas rápida la convergencia, lo único que cambia para este caso es que no se actualizan los parámetros en cada iteración si no que se inicializan al principio de forma diferente a la anterior, y mantienen el mismo valor durante todo el proceso.

- Por último se encuentra la zona de liberar memoria y devolver resultado.

La estructura de las funciones *kernel*, para la actualización Primal Dual es la siguiente

```
1  __global__ void update_primalOdual_GPU(...){
2
3      int i = blockDim.x*blockIdx.x + threadIdx.x;
4      int j = blockDim.y*blockIdx.y + threadIdx.y;
5
6      if( i<width && j<height ){
7      //Coordenadas en memoria de textura
8          float tx=i+0.5f;
9          float ty=j+0.5f;
10         /*
11         Realizacion de los calculos
12         */
13     }
14 }
```

En primer lugar lo que hace es averiguar para que pixel se esta ejecutando el *kernel*, una vez hecho esto se comprueba que ese supuesto pixel esta dentro de la imagen, ya que cabe la posibilidad que se hayan ejecutado mas hilos que píxeles tiene la imagen. Tras esto calcula las coordenadas tx y ty, que serán las coordenadas donde leerá en la memoria de textura, esto es porque la memoria textura lee la imagen como algo constante, y no discreto como la memoria global, por lo que realiza una interpolación, y al sumarle ese 0.5f se desplaza al centro del pixel en cuestión pudiendo así comprobar realmente el valor deseado.

En cuanto a los cálculos para cada uno de los *kernel* de cada uno de los métodos es literalmente programar los resultados de las deducciones matemáticas que se encuentran en el capítulo de denoising o de fusión en la memoria.

Apéndice D

Interfaces

D.1. Interfaz de prueba de parámetros

El origen de esta interfaz fue improvisado, ya que no se pensaba hacer de antemano, pero surgió de la necesidad, ya que en los artículos en los que se basaba el proyecto eran algo ambiguos con el tema de los parámetros que había que introducir.

D.1.1. Requisitos

En cuanto a requisitos, se definieron sencillamente los siguientes:

- Se introducirá una imagen desde archivo para las pruebas, que será mostrada en la interfaz.
- Existirá un apartado para introducir ruido en la imagen, una vez introducido, se deberá mostrar en la interfaz.
- Se tendrá la opción de elegir entre cada uno de los algoritmos de denoising del proyecto.
- Según el algoritmo elegido se mostrará la posibilidad de hacer unas pruebas u otras.
- Se podrán seleccionar los rangos en los que hacer las pruebas y el valor de otros parámetros que afecten al algoritmo aunque la prueba no se centre en ellos.
- Se mostrarán los resultados en una gráfica embebida en la interfaz.
- Se mostrará también al final de la prueba la imagen resultado.

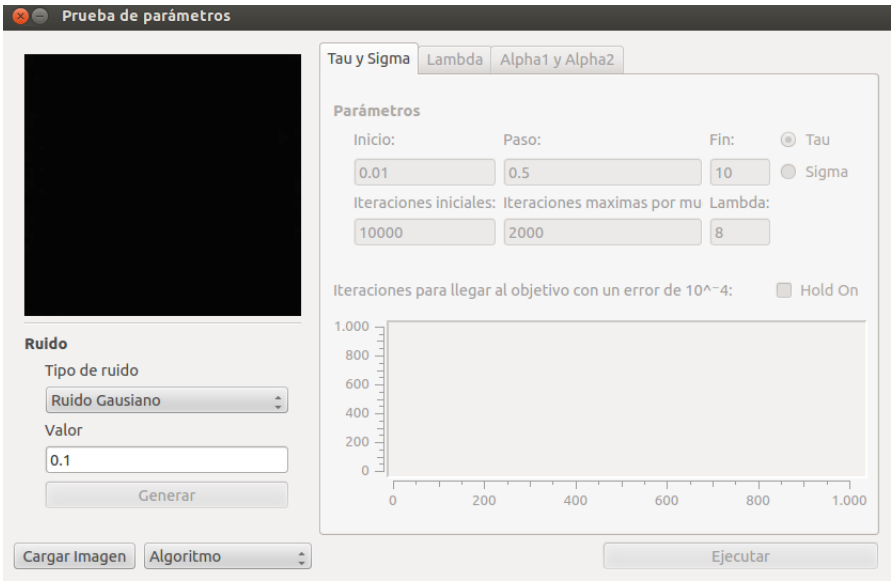


Figura D.1: Aspecto de la interfaz de prueba de parámetros al inicio de su ejecución.

- Las pruebas se realizaran en un thread aparte para que la interfaz no quede bloqueada durante la prueba.
- Se mostrará con una barra de carga o similar el proceso de la prueba para tener noción del tiempo que va a necesitar.

D.1.2. Resultado de la interfaz

Se muestra a continuación el diseño de la interfaz, por un lado, el estado inicial en la figura D.1, y por otro un esquema de la interfaz completa, con todos los elementos visibles para facilitar la explicación en la figura D.2.

1. Se trata de el *widget* que mostrará en cada momento la imagen con la que se este trabajando, al principio mostrará la imagen que se cargue, y una vez introducido el ruido, mostrará la imagen ruidosa.
2. Es la zona de introducir ruido en la imagen, en ella se puede elegir el tipo de ruido que se desea mediante el *combobox*, así como su valor.
3. En esta zona simplemente está el botón para cargar la imagen desde archivo, y el *comboBox* para la elección del algoritmo sobre el que se quiere realizar la prueba.
4. Esta es la zona que más espacio ocupa en la interfaz, pero también es la más importante. En ella aparecen dos zonas diferenciadas.

Figura D.2: Esquema de todos los elementos de la interfaz de parámetros.

a) Se trata de la zona de introducción de los intervalos a muestrear de las distintas variables así como, por un lado parámetros relacionados con las pruebas, y por otro parámetros relacionados con el método. Más adelante se detallarán todos ellos.

b) En esta zona representan los resultados que se han obtenido en la prueba, cabe destacar que para la mayoría de las pruebas esta representación es en tiempo real gracias a que la interfaz y las pruebas se ejecutan *threads* independientes.

También existe en esta zona la opción de hacer *HoldOn*, es decir, la opción de que se conserven los datos en la gráfica de una prueba, al realizar la siguiente, útil para poder comparar los resultados con otros métodos, o elegir otro rango de acción de la prueba sin que tengas que repetir parte de ella.

5. Se trata simplemente del botón de ejecutar que desata todas las pruebas.

6. Esta zona solo se muestra después de que se haya realizado una prueba, en ella se muestra el antes y el después de la imagen ruidosa, siendo el después el mejor resultado conseguido por todas las muestras ejecutadas en la prueba. El aspecto de esta sencilla ventana emergente se muestra en la figura D.3.

7. Se trata de la barra de carga, calcula el número de muestras procesadas para dar información sobre el estado de toda la prueba.

Figura D.3: Aspecto de la ventana emergente que se genera tras pulsar el botón mostrar resultado después de ejecutar una prueba.

D.1.3. Detalles de utilidad

Se van a detallar a continuación que pruebas se pueden realizar para cada algoritmo, no de forma cualitativa como se ha hecho en la memoria, sino que se van a describir los parámetros requeridos para cada prueba así como los resultados que genera.

Prueba τ y σ

Los parámetros que se requieren para realizar esta prueba, son, por un lado el rango de valores de τ o σ en los que se va a realizar la prueba y por otro lado los parámetros relacionados con las prueba, como el número de iteraciones a realizar al principio para generar una solución que haya convergido y las iteraciones máximas a realizar a lo largo de la prueba, ya que si este número no está acotado, para una mala elección de los parámetros puede tardar mucho a converger y la prueba se eternizaría.

Por último hay que introducir también el valor de λ, lo que ofrece un mayor rango de posibilidades en las pruebas. Cabe destacar que la interfaz es única para pruebas de τ y de σ, por lo que existe un *radioButton* que da la opción de elegir una u otra, tampoco sería necesario que hubiera dos pruebas diferenciadas, ya que una depende de la otra.

En este caso solo se mide la velocidad de convergencia por lo que no existe una imagen resultado mejor que las demás por lo que en la ventana de mostrar el resultado se mostrará la solución sobreiterada.

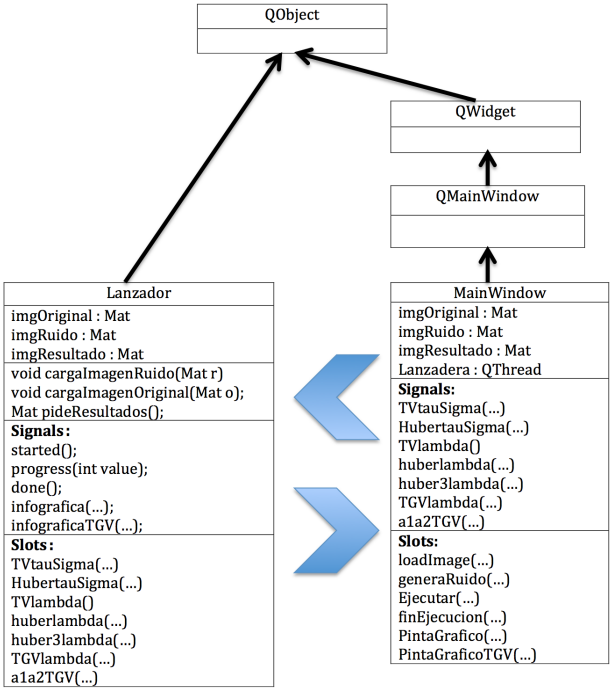


Figura D.4: Pseudodiagrama de clases de parte de la aplicación de prueba de parámetros.

Prueba de λ

Esta prueba está disponible para todos los algoritmos, es bastante simple, sólo es necesario introducir el intervalo de datos que se desean probar, y las iteraciones que se han de realizar para cada muestra.

Prueba de α_1 y α_2

Esta prueba es exclusiva del algoritmo TGV, es la mas compleja de las realizadas ya que estas dos variables no dependen de sí mismas, por lo que el número de muestras se multiplica.

Para esta prueba hay que introducir los intervalos en los que realizar la prueba, así como las iteraciones a realizar por muestra, y el valor de λ.

El resultado de esta prueba es distinto a todos los anteriores ya que ahora el valor del análisis señal ruido a realizar depende de dos variables, por lo que el resultado se muestra en un espectrograma.

D.1.4. Detalles de implementación

En este apartado se va a detallar lo más representativo de la parte de la implementación tanto lo relacionado con la interfaz en *QT* como la parte de las pruebas en *C++*.

En toda la aplicación se pueden diferenciar tres clases principales, una es la que soporta la interfaz (*MainWindow*), otra se utilizará para ejecutar todas las pruebas (*Lanzador*), y otra en la que están programados los distintos algoritmos de denoising (*Normas*). En la figura D.4 se muestra un pseudodiagrama de clases en el que se muestra la jerarquía de las dos primeras clases.

Se observa que todo esta heredado de la clase *QObject* que es la principal clase de *QT*, gracias a ella por ejemplo se pueden hacer uso de los signals y slots.

En este punto el diagrama se divide en dos ramas, una llega ya a la clase lanzador, y otra a *QWidget*, esta última clase será la que dote a sus hijos de distintas propiedades para su representación en la interfaz.

Tras ella se encuentra la clase *QMainWindow* que simplemente es una distribución predefinida en la interfaz, con posibilidad de crear de forma sencilla barras de herramientas, menús, barras de estado, etc.

Por último al final de esta rama se encuentra la clase *MainWindow*, esta es la propia interfaz rellena ya con todos los elementos.

A continuación se detalla las distintas clases principales nombradas anteriormente.

MainWindow

Es la clase principal de la aplicación maneja todo lo relacionado con la interfaz, esto queda claro al observar los distintos slots que la componen. Solo con leer el nombre se observa que cada uno de ellos atiende a una de las posibles acciones que se pueden desatar en la interfaz, cargar una imagen (*loadImage()*), introducirle el ruido (*gereraRuido()*, ejecutar las pruebas (*Ejecutar()*), etc. Uno de los requisitos más laboriosos fue el hecho de que las pruebas se ejecutaran en otro thread para que la interfaz no quedara bloqueada durante ese tiempo, para ello se construyó la clase *Lanzador*, que esta asociada a la variable lanzadera de clase *QThread*. El funcionamiento de este sistema es el siguiente:

```
1 lanzadera=new QThread();
2 lanzador.moveToThread(lanzadera);
```

Se declara el Thread y se indica que la instancia de la clase *Lanzador* se ejecute en él.

```
connect(&lanzador, SIGNAL(done()), this, SLOT(finEjecucion()));
connect(this, SIGNAL(TVtauSigma(float, float, float, int, int, int,
    float)), &lanzador, SLOT(TVtauSigma(float, float, float, int, int,
     int, float)));
```

En las líneas anteriores se indica que cuando la clase lanzador emita la señal *done*, es decir, a finalizado las pruebas, se ejecute el slot $finEjecucion()$. La segunda línea le indica que cuando la interfaz emita la señal $TVtausigma$ con todos los parámetros, en el lanzador se ejecutará el slot con el mismo nombre, quien realizará la prueba. Esta segunda línea se repite para todas las pruebas que se puede realizar. Todas estas interaciones entre las clases $Mainwindow$ y $Lanzador$, están representadas en el diagrama por las flechas azules.

```
lanzadera->start();
emit TVtauSigma(/* parametros de la prueba */);
```

Ahora en el momento que quiere ejecutar una prueba en concreto se inicializa el thread y se emite la señal que capturará la el lanzador, quien ejecutara la prueba, además lo hará en un thread paralelo, por lo que la interfaz no quedará bloqueada esperando el cálculo.

Una vez realizada la prueba y leídos los resultados se ejecutará lo siguiente.

```
    lanzadera->quit();
```

Con lo que se detendrá el thread a la espera de la siguiente ejecución.

Al igual la interfaz emite señales con los valores de una prueba para que la realice el lanzador, este devuelve los valores de la prueba, estas señales serán detalladas en el siguiente apartado donde se define la clase Lanzador.

Dentro de la interfaz también cabe destacar la inclusión de *widgets*[1] de la librería *qwt*, que se trata de una librería que incluye nuevos componentes y utilidades para aplicaciones técnicas, en este caso han sido usados dos de estos *widgets*, ambos para la representación de datos, uno más sencillo el *qwtPlot* que se encarga simplemente de representar en una gráfica 2D los datos generados en las pruebas, y otro más complejo que realiza una representación 3D, mediante un espectrograma, útil para la prueba de α_1 y α_2.

[1]Widget: Dícese de cada uno de los elementos de la interfaz

Para este último *widget* el espectrograma ha habido que reprogramar una parte de la propia clase, ya que estaba preparado para representar funciones continuas y no datos discretos, como se deseaba en este caso, además se introdujo en esta clase una función para realizar la interpolación de los datos, de esta forma el resultado luce mucho más vistoso.

En este problema el dato de entrada era una matriz de datos con el resultado de las pruebas y distintos valores del rango de datos, inicio, paso y final, de las muestras de cada una de las variables. La función en cuestión se ejecutaba una vez para cada pixel de la gráfica que representaba por lo que tenía dos parámetros x e y de entrada. Para cada ejecución se realizaban los siguientes cálculos para la interpolación.

1. Cambio de coordenadas a el rango de resultados:

$$x_g = x - min_x$$

$$y_g = y - min_y$$

2. Cálculo del indice del voxel[2] correspondiente:

$$i_x = floor(\frac{x_g}{paso_x})$$

$$i_y = floor(\frac{y_g}{paso_y})$$

   Siendo $floor$ un redondeo a la baja.

3. Ahora se transforma el índice a la esquina inferior izquierda, mediante el siguiente código.

```
1 if(xg<(ix+0.5)*pasox) ix=ix-1;
2 if(yg<(iy+0.5)*pasoy) iy=iy-1;
```

4. Se calculan ahora las celdas para interpolar.

```
1 int ix0, ix1, iy0, iy1;
2 if(ix<0) ix0=ix+1;
3 else ix0=ix;
4 if(iy<0) iy0=iy+1;
5 else iy0=iy;
```

[2]posición en la matriz de datos

```
6 if(ix+1>columnas) ix1=columnas;
7 else ix1=ix+1;
8 if(iy+1>filas) iy1=filas;
9 else iy1=iy+1;
```

5. se calcula la distancia normalizada.

$$d_x = \frac{x_g - (i_{x0} + 0{,}5)paso_x}{paso_x}$$

$$d_y = \frac{y_g - (i_{y0} + 0{,}5)paso_y}{paso_y}$$

6. Por último se consultan los valores de los indices obtenidos en la matriz de datos, y se realiza la interpolación.

```
1 float V00=datos[ix0+iy0*columnas];
2 float V10=datos[ix1+iy0*columnas];
3 float V01=datos[ix0+iy1*columnas];
4 float V11=datos[ix1+iy1*columnas];
5 return V00*(1-dx)*(1-dy)+V10*dx*(1-dy)+V01*(1-dx)*dy+V11*dx*dy;
```

Ese valor devuelto será representado en la gráfica en el pixel correspondiente.

Lanzador

Esta es la clase que ejecuta las pruebas que le ordena la interfaz, no ella directamente sino que llama a otra clase *Normas*, que es la que ejecuta los algoritmos de denoisng. El porqué de esta división en dos clases para hacer una sola tarea, es porque la clase *Normas* como se verá esta en *Cuda*, y *Qt* no permitía heredar de *QObject* en un fichero *.cuh*, por lo que no había posibilidad de capturar señales ni emitirlas desde esa clase, obligando ejecutarla directamente en el thread principal.

La clase *Lanzador* entra en juego cuando la interfaz emite una señal para ejecutar una prueba, en ese momento se ejecuta un slot de esta clase. En el diagrama D.4 se observa que para cada señal que puede lanzar la interfaz, existe un slot del mismo nombre que captura esta señal y realiza la prueba.

Estas pruebas como se ha visto en apartados anteriores son principalmente de dos tipos, de convergencia o de análisis señal ruido, a continuación se muestra el esqueleto de ambos tipos de funciones.

```
1  //Primero se calcula la imagen sobre iterada
2    Mat referencia = TV(/*Parametros*/);
3    for (/*Todas las muestras*/){
4        //Emite el progreso
5        emit progress(/*Porcentaje*/);
6        //Ejecuta la prueba
7        TV(/*Pametros*/);
8        //Emite una senal con el SNR resultante
9        emit infografica(/*Resultados*/);
10   }
11   //Senal de fin del programa
12   emit done();
```

El fragmento de código anterior se trata del esqueleto de las funciones que ejecutan las pruebas con τ y σ, es útil para ver el el funcionamiento de los distintos tipos de señales y slots que entran en juego en la comunicación entre las clases *MainWindow* y *Lanzador*.

Una vez la interfaz emite la señal para que se ejecute una prueba con sus correspondientes parámetros queda a la espera de recibir información por parte del *Lanzador* sobre el estado del proceso. Durante el bucle de procesamiento de las muestras este envía dos señales, una para que según la iteración se actualice la barra de progreso de la interfaz, y otra que va devolviendo los resultados generados para las distintas pruebas.

Por último se emite la señal *done*() que simplemente indica a la interfaz que la prueba a finalizado para que esta detenga el thread y quede dispuesta para la realización de otra prueba si se requiriera.

En cuanto al esqueleto de la función que ejecuta las pruebas de los parámetros de ponderación que se muestra a continuación. Es similar en el tratamiento de las señales, el único cambio es que no realiza una prueba sobreiterada inicial, ya que en este método no es necesario y que en cada iteración realiza una comprobación del resultado ya que tiene que almacenar la imagen que genera mayor índice señal ruido.

```
1
2    for (/*Todas las muestras*/){
3        //Emite el progreso
4        emit progress(/*Porcentaje*/);
5        //Ejecuta la prueba
6        TV(/*Pametros*/);
7        if(SNR>SNR_ant){
8            //Actualiza el resultado
9        }
```

```
10          //Emite una senal con el SNR resultante
11          emit infografica(/*Resultados*/);
12      }
13      //Senal de fin del programa
14      emit done();
```

Normas

En cuanto a la clase normas simplemente reúne todos los algoritmos de denoising programados, con la novedad en este caso de para las pruebas de convergencia para cada iteración va evaluando el resultado, y para las pruebas de los parámetros de ponderación realiza una análisis señal ruido del resultado al final de todas las iteraciones.

D.2. Interfaz de visualización

El objetivo de estas interfaz es principalmente ser el soporte para los problemas de denoising programados a lo largo del proyecto, a su vez también es útil para realizar pruebas sobre el resultado que generan distintas combinaciones de parámetros, no realiza esto de forma automática y cuantitativa como la interfaz de parámetros explicada en el capítulo anterior sino que es más selectiva, da la posibilidad de ver el resultado de cada prueba, y decidir unos parámetros según el criterio visual y no simplemente por el valor generado por en análisis señal ruido.

D.2.1. Requisitos

Se detallan a continuación una serie de requisitos que se decidieron de la aplicación a modo de análisis previo.

- Existirán dos modos de ejecución, uno en el que se introduzca una imagen desde archivo para realizar la prueba, y otro que muestre si es posible la cámara web del ordenador para realizar el denoising.
- Existirá como en el caso de la interfaz de parámetros una zona para seleccionar e introducir el ruido que se desee.
- Para cada uno de los algoritmos programados existirá una zona en la que se muestren los posibles parámetros de cada uno para poder realizar las distintas pruebas

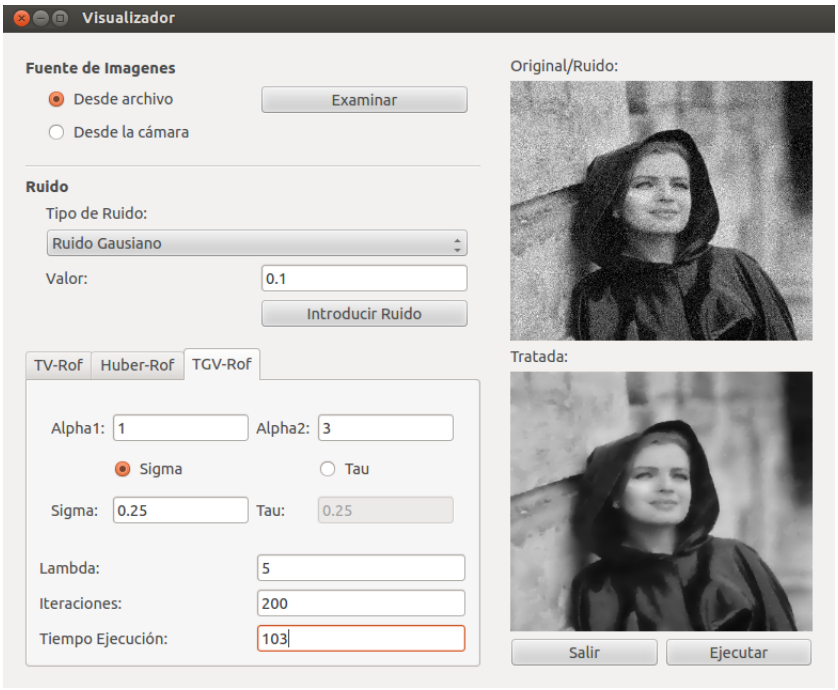


Figura D.5: Aspecto final de la aplicación para visualizar los resultados del denosing.

- Se mostrará en todo momento el antes y el después del denoising.
- Para cada ejecución del denoising sean cual sean las opciones elegidas devolverá el tiempo de ejecución.
- La interfaz deberá estar totalmente operativa mientras se realiza el denoising, por lo que se tendrán que programar distintos threads

D.2.2. Resultado de la interfaz

En la figura D.5 se observa el aspecto final de la interfaz después de haber realizado una prueba.

En la figura D.6, se observa cada una de las zonas que cumplen los distintos requisitos en lo que utilidad se refiere.

1. Esta es la zona de elección de fuente de imágenes, existen dos posibilidades, cargar una imagen desde archivo con su correspondiente botón para elegir la imagen, o cargar las imágenes de la webcam del ordenador. Una vez pulsada esta última opción se comienzan a mostrar las imágenes capturadas en la zona correspondiente.

2. Se trata de la zona de introducción de ruido en la imagen, es similar a la interfaz de parámetros, por un lado se elige el tipo de ruido, y por otro la cantidad.

3. Es la zona de introducción de parámetros para cada uno de los tres métodos, y donde muestra el tiempo invertido en la ejecución.

   a) Esta es la zona donde se introducen los parámetros exclusivos para cada método, es decir α_1 y α_2 para TGV, y α para $Huber$, además en este último método se seleccionará el algoritmo con el que se desea realizar la ejecución, ya que como se ha dicho en repetidas ocasiones $Huber$ admite una variación del algoritmo de minimización que acelera la convergencia.

   b) Aquí se muestran los parámetros de convergencia, es decir, τ y σ en caso de que proceda.

   c) En este punto se introduce λ.

   d) Caja de texto para introducir el número de iteraciones a ejecutar.

   e) Por último en esta zona devuelve el resultado en milisegundos del tiempo invertido para el denoising.

4. En esta zona se muestra en primer lugar la imagen sin ruido, según la introduces desde archivo, o la lee desde la cámara, y la imagen ruidosa, una vez se haya seleccionado el tipo y el nivel de ruido.

5. Muestra la imagen resultante de ejecutar el denoising para los parámetros seleccionados en el panel de la izquierda (3).

6. En esta zona simplemente están ubicados los botones de Salir y Ejecutar, cuya función es evidente.

D.2.3. Detalles de implementación

Una de las principales dificultades que se encontró al hacer la aplicación fue cumplir el requisito que dice que la interfaz ha de quedar operativa mientras se ejecuta el denosing.

A priori se pensó en hacer dos threads como para la interfaz de parámetros, uno para la interfaz y otro para que ejecutara el método propiamente dicho, pero hubo que contar también con el hecho de que existía la posibilidad de que las imágenes a recuperar vinieran desde la cámara por lo que hubo que realizar un tercer thread para la captura de estas.

Figura D.6: Esquema de las distintas zonas de la interfaz de visualización.

Aunque pudieran parecer iguales el funcionamiento de estos dos threads es muy distinto, para entenderlo hay que observar la figura D.7.

Como se puede observa la estructura es muy parecida a la de la aplicación de prueba de parámetros con el añadido del thread de lectura de la cámara.

Antes de la implementación de este thread, se barajaron dos posibilidades.

- Por un lado, se barajó el hecho de hacerlo de igual modo que la clase *Lazador*, es decir, que la interfaz cada cierto tiempo ejecutaría el thread para conseguir un frame de vídeo y trabajar con él.

  El principal defecto de esta técnica es que se ha de inicializar el thread a cada frame que pide la interfaz, y esto sucede con mucha frecuencia unas 30 veces por segundo.

- La otra opción fue el programar una clase que heredara directamente de *QThread*, y que se ejecutara cada vez que la aplicación se pusiera en modo vídeo, y durante este tiempo lea frames de la cámara.

Esta última fue la opción elegida como se puede observar en la figura D.7, su funcionamiento en resumen consiste en que cada vez que el usuario indica el modo vídeo se activa el thread de la siguiente forma.

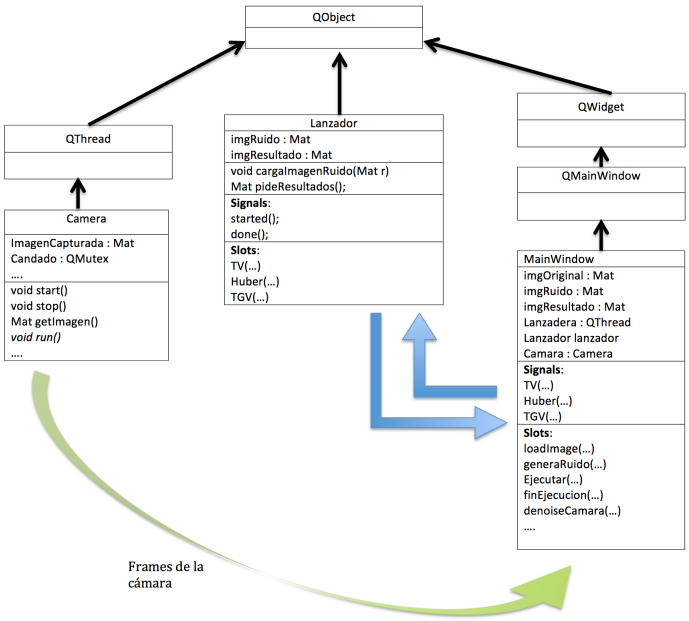


Figura D.7: Pseudodiagrama de clases de la aplicación de visualización

```
camera.start();

timer = new QTimer(this);
connect(timer,SIGNAL(timeout()), this, SLOT(leerCamara()));
timer->start(15); // se puede poner 0
```

En primer lugar activa el thread simplemente llamando la función *start()*, tras esto se activa un *timeout* que leerá una imagen del thread cada 15 milisegundos en este caso. Esta lectura se realiza de la siguiente forma.

```
void MainWindow::leerCamara(){
  Candado.lock();
    imgOriginal = camera.getImagen();
  Candado.unlock();
    imgRuido=addNoise(imgOriginal,0.0,valorRuido);
    ui->mostrar_original->setImage(imgRuido);
    ui->mostrar_original->updateGL();
}
```

Se llama a la función *getImagen()*, que devuelve la imagen que en ese momento

captura la cámara.

Una vez capturada cuando el usuario pulsa ejecutar se realiza el denosing de esa imagen, si al finalizar la ejecución no se ha salido del modo vídeo, se vuelve a realizar el denoising de la última imagen que la interfaz haya leído del thread de captura.

Una vez se haya salido del modo de vídeo, por ejemplo para cargar una imagen desde archivo, o directamente para cerrar la aplicación, se detiene la ejecución del thread de captura de la siguiente forma.

```
1 timerDenoising->stop();
2 timer->stop();
3 camera.stop();
```

Además de detener el thread se detiene el timeout que se había activado para la captura de las imágenes, por lo que la interfaz no pedirá al thread más imágenes, lo que produciría un error.

A continuación se resume la implementación de las distintas clases implicadas.

MainWindow

Poco mas se puede de decir de esta clase, la única miga que pudiera tener es el tratamiento de los distintos threads, de la que ya se ha hablado. Más allá de eso su estructura y funcionamiento de esta clase es similar que en la aplicación de prueba de parámetros.

Lanzador

El funcionamiento de esta clase es similar que en la aplicación de prueba de parámetros.

Capture

En cuanto a la implementación de la clase *Capture*, es una clase que hereda de *QThread*, por lo que tiene unas propiedades especiales, su parte central se encuentra en la función *run*(), que es la que se ejecuta automáticamente cuando el *thread* se inicia. Se muestra a continuación esta función *run()* de forma simplificada.

```
1 cv::VideoCapture capture(0);
2 while (/*Fin del thread*/){
3   capture >> frame;
```

```
4    cv::cvtColor(frame,imGrey,CV_RGB2GRAY);
5    Candado.lock();
6    imagenParaInterfaz=imGrey;
7    Candado.unlock();
8  }
```

Al principio inicia el canal de captura, tras ello comienza el bucle de captura, dentro de él captura un frame, convierte a escala de grises, y almacena el resultado para que la interfaz lo lea cuando lo desee. Para evitar que la interfaz lea la imagen justo cuando esta clase la esta escribiendo, se ha colocado un *mutex* en el momento en el escribe la imagen, y cuando la lee en la interfaz.

Normas

Esta clase es sencillamente una reunión de todos los métodos de denosing del proyecto. Para mas información acerca de la implementación de esta clase diríjase al anexo correspondiente.

D.3. Interfaz de fusión

Esta interfaz es el soporte de todo el proceso de fusión de mapas explicado en la memoria, desde la elección de los distintos mapas de profundidad y elección del ruido pasando por la virtualización hasta la realización de la fusión.

D.3.1. Requisitos

A continuación se muestran una serie de requisitos generados antes de realizar la aplicación, con directrices de su funcionamiento.

- Se deberán introducir los mapas profundidad en la aplicación y simultáneamente las posiciones y parámetros de las cámaras de referencia.
- Será posible la visualización de cada uno de los mapas una vez introducidos.
- Se deberá de poder introducir ruido gausiano así como datos espurios a cada mapa de profundidad por separado.
- En todo momento se tendrá la posibilidad de utilizar para la virtualización y posterior fusión los mapas estimados, o los mapas perfectos (ground truth) indistintamente.

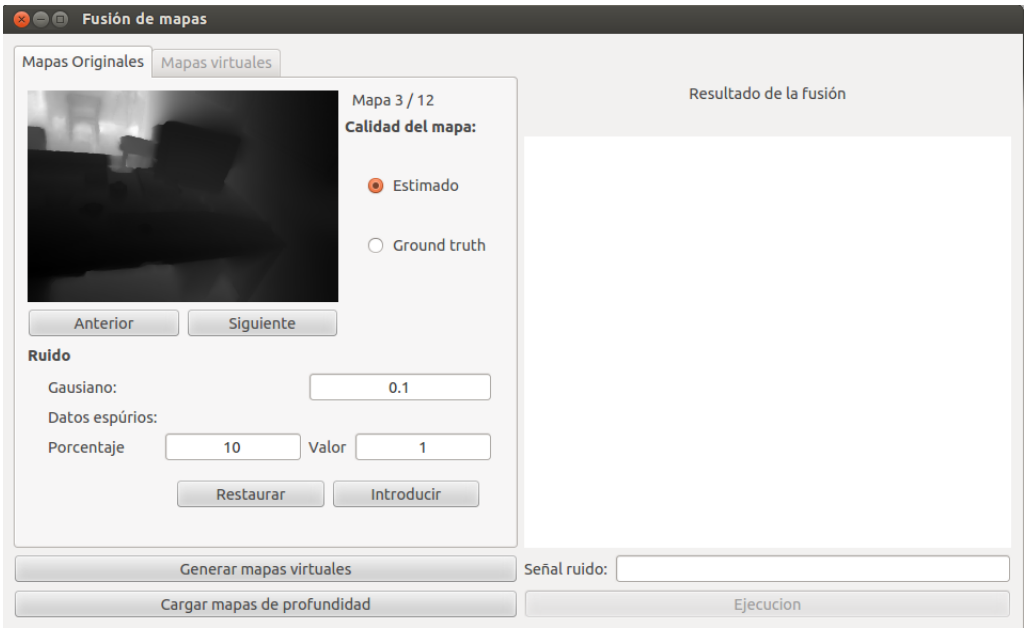


Figura D.8: Aspecto inicial de la interfaz de fusión habiendo cargado ya un lote de mapas de profundidad

- Una vez haya mapas cargados se habilitará la opción de realizar la virtualización.
- Antes de realizar la virtualización se elegirán las resoluciones de los distintos ejes del cubo.
- Una vez virtualizados se podrán consultar los mapas resultantes.
- Una vez realizada la fusión se mostrará el resultado así como el resultado del análisis señal ruido respecto del mapa en ground truth visto desde la posición sobre la que se encuentra el resultado.
- La fusión de los mapas se realizará en un thread independiente de modo que la interfaz no quede bloqueada durante este proceso.

D.3.2. Resultado de la interfaz

En la figura D.8 se observa el aspecto inicial de la aplicación una vez cargados un lote de mapas de profundidad, y en la figura D.9 la situación final después de haber realizado la virtualización y posterior fusión de los mapas de profundidad.

Como en las anteriores ocasiones la figura D.10 muestra numeradas cada una de las partes en las que se divide la interfaz para facilitar la explicación.

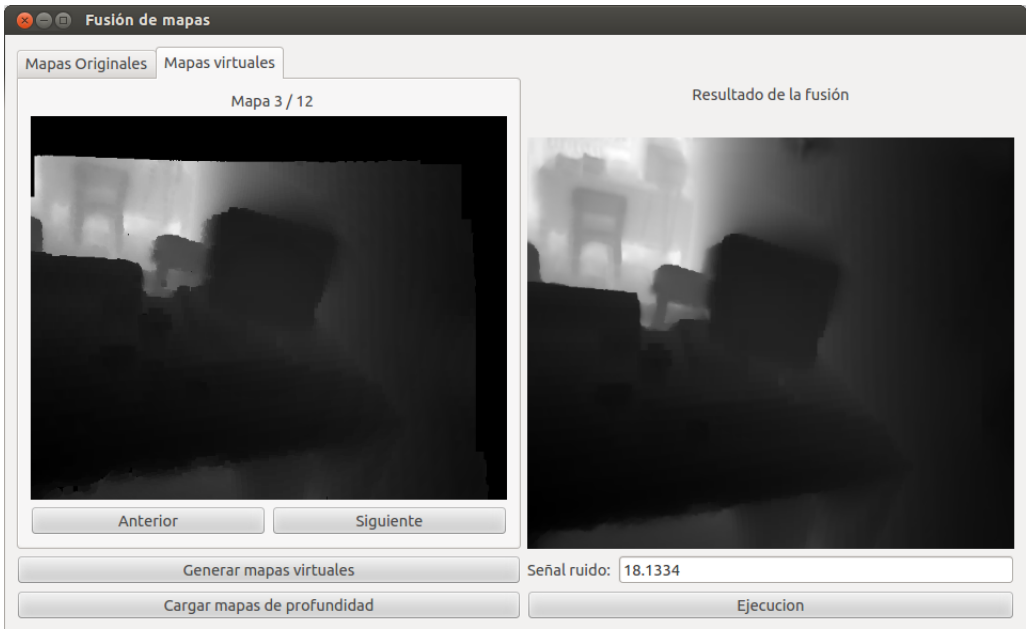


Figura D.9: Aspecto final de la interfaz de fusión habiendo realizado ya la fusión de los mapas de profundidad virtualizados

1. Se trata de los botones para cargar los mapas de profundidad al comienzo de la ejecución y para generar los mapas virtuales en el momento en el que es posible.

2. Esta zona es la de elección del tipo de los mapas así como de introducción de ruido a estos, que serán la entrada del siguiente paso.

   a) Aquí se muestran todos lo mapas introducidos al comienzo, es posible navegar entre ellos para poder visualizarlos todos mediante los botones anterior y siguiente.

   Además en esta parte existe la opción de usar mapas en ground truth o mapas estimados, esta elección se aplicará solamente al mapa que se este mostrando en ese momento.

   b) Se trata de la zona de generación de ruido en los mapas de profundidad introducidos al inicio, las opciones disponibles son la introducción, por un lado de ruido gausiano, y por otro de datos espurios, para lo que se indica el porcentaje de pixeles afectados y la variación máxima en estos. Como en el caso anterior estas modificaciones se ejecutan solamente en el mapa que se muestra en ese momento.

3. Es la pestaña de visualización de los mapas de profundidad ya colocados

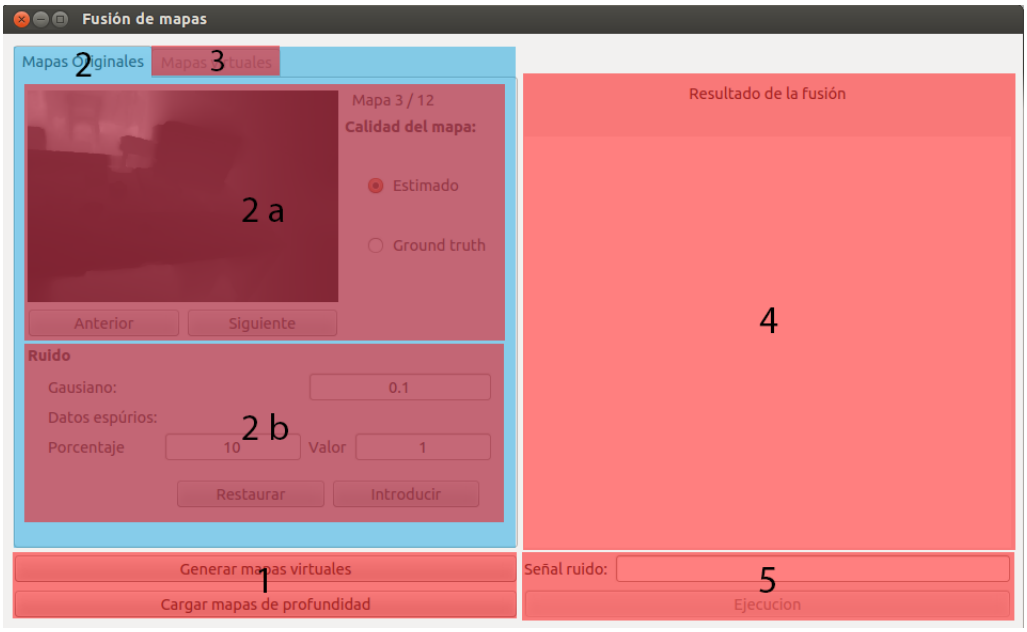


Figura D.10: Esquema de las distintas partes que conforman la interfaz de la fusión de mapas.

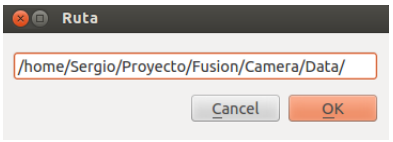


Figura D.11: Diálogo para la introducción de la ruta de los mapas de profundidad

todos respecto de la misma referencia, esta parte se muestra en la figura D.9. Como en el caso del visualizador de mapas inicial da la opción de navegar por todos ellos mediante los botones anterior y siguiente.

4. Se trata de la zona donde se mostrará el resultado final del experimento.

5. Por último, en esta zona se encuentra, por un lado el botón que ordenará la ejecución de la fusión siempre y cuando sea posible, y por otro la caja de texto donde se devolverá el valor del análisis señal ruido del resultado obtenido respecto de el ground truth correspondiente a la cámara de referencia del resultado.

Existen partes de la interfaz que no se muestran en la figura D.10, ya que se tratan de las distintas ventadas emergentes que se muestran para que el usuario introduzca los datos oportunos.

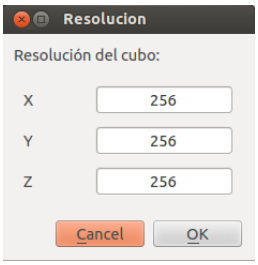


Figura D.12: Diálogo para la introducción de la resolución en las distintas dimensiones del cubo

Figura D.13: Diálogo de introducción de parámetros para la fusión de mapas de profundidad.

En la figura D.11, se muestra la primera de ellas, se acciona cuando el usuario pulsa el botón de cargar los mapas, en ella se debe introducir la ruta donde se almacenan los mapas de profundidad de entrada, así como los parámetros y posición de las cámaras desde las que fueron construidos.

El diálogo de la figura D.12, se muestra antes de realizar la virtualización de los mapas de entrada, una vez se a pulsado el botón correspondiente, en él se introduce la resolución del cubo en las distintas dimensiones.

Por último, el diálogo mostrado en la figura A.5 entra en acción justo antes de realizar la fusión, en él el usuario tiene a posibilidad de introducir nuevos parámetros para el método que generará el resultado final.

D.3.3. Detalles de implementación

En cuanto a la implementación la estructura es similar a las interfaces anteriores en cuanto al uso de threads, en este caso, como en la interfaz de prueba de parámetros solo existen dos threads, el de la propia interfaz, y el que se encarga de realizar la fusión, y su estructura es la misma que en caso anterior, en la figura D.14 se observa esta distribución.

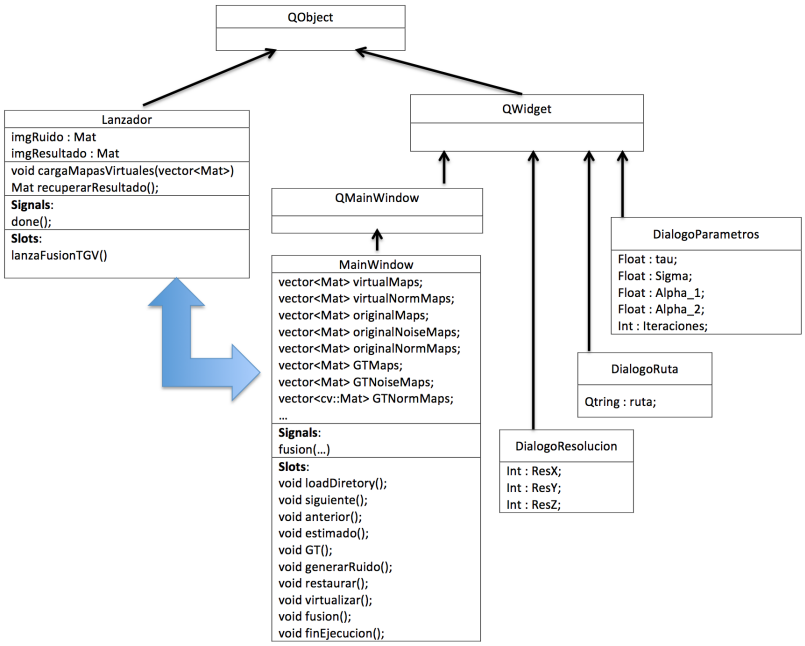


Figura D.14: Pseudodiagrama de clases de la interfaz de Fusión

MainWindow

En cuanto a la clase *MainWindow* es similar a las anteriores ocasiones. Se basa en una serie de slots que se actúan según las señales emitidas por los elementos de la interfaz.

En cuanto a los parámetros almacenados en la clase, en otras ocasiones se almacenaban las distintas imágenes del proceso, la original, la ruidosa, etc.

En este caso esas imágenes se han convertido en vectores de imágenes, además no hay una entrada como en los problemas de denoising, hay dos, por un lado están los mapas de profundidad estimados, y por otro están los ground truth correspondientes, por lo que se duplican las variables, además de eso, hay que que tener en cuenta que para representar los mapas han de estar normalizados, lo que significa otro vector más. Por último también almacena el resultado de la virtualización normalizado y sin normalizar.

Lanzador

El lanzador es incluso más sencillo que en los casos anteriores ya que solo existe una función a la que llamar, se trata la que hace la fusión. Esta función tiene como entrada una serie de parámetros que se le envían al lanzador mediante la señal*fusion* generada por la interfaz, además de eso necesita, como es lógico los

mapas ya virtualizados, para ello la interfaz antes de emitir la señal, ejecutará la función *cargaMapasVirtuales()*, de este modo ya se podrá realizar la fusión y en este caso se realizará en un thread paralelo, con lo que la interfaz no se quedará inutilizada mientras tanto.

Diálogos

En cuanto a las distintas clases de diálogos su funcionamiento en bien sencillo. Se ejecutan desde la clase *MainWindow*, una vez aceptados la interfaz consulta los datos que ha introducido el usuario mediante las distintas funciones *get* que existen en las distintas clases diálogo.

Programación en GPU

Hasta este punto se ha detallado el funcionamiento de la interfaz, pero lo realmente importante es lo que queda detrás de ella, se trata de las distintas clases que realizan la virtualización de los mapas y la fusión de los mismos.

La programación de la creación de los mapas virtuales esta detallada en el anexo correspondiente a la fusión. En cuanto a la fusión propiamente dicha, se ha creado una clase llamada *tgvFusion* que simplemente tiene implementado el método de forma similar a la clase $Normas$ utilizada para el denoising.

Apéndice E

Fusión

E.1. Mapas virtuales

La generación de mapas virtuales a partir de otros es un punto importante del proceso ya que para para realizar la fusión, todos los mapas implicados han de estar colocados respecto a la misma referencia.

Todos los datos de entrada a este proceso se basan en los mapas de profundidad y su información asociada.

- La matriz K de información sobre los parámetros de la cámara.
- La matriz de transformación que da información de la situación de esa cámara en el espacio.
- El mapa de profundidad en cuestión, que se trata como se ha dicho en otras ocasiones de una imagen que almacena para cada pixel la distancia de la cámara al punto concreto de la escena.

Para conseguir la transformación de los mapas se han de realizar varias transformaciones y operaciones que se detallan a continuación, mas tarde se definirá la aplicación de cada una de ellas en el proceso.

Proyección de 3D a 2D

Esta operación proyecta un punto cualquiera del espacio en la imagen, de este modo se sabrá a que pixel del mapa de profundidad corresponde ese punto del espacio.

La fórmula que hay que aplicar es la siguiente.

$$u = f_x \frac{x}{z} + c_x$$

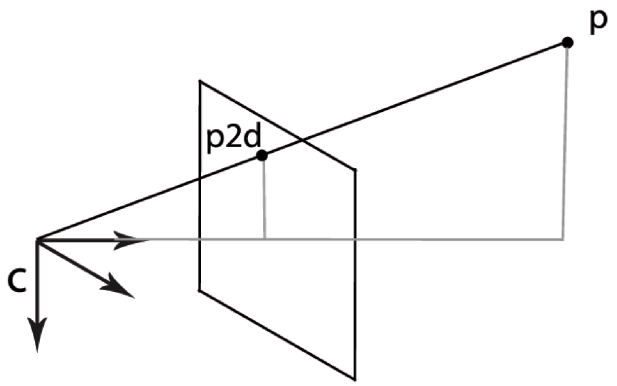


Figura E.1: Esquema de actuación del método para construir mapas de profundidad a través de imágenes

$$v = f_y \frac{y}{z} + c_y$$

Siendo u y v las coordenadas del mapa $(p2d)$ a las que corresponde al punto $p = [x, y, z]$, f_x y f_y las distancias focales y ,c_x y c_y parámetros de la cámara como las distancias focales, y por tanto datos de entrada en el programa.

Proyección de 2D a 3D (Back proyection)

Es la operación contraria que la anterior, devuelve la posición en el espacio de de un punto del depth map, la expresión que se ha de aplicar en este caso es la siguiente.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \frac{u-c_x}{f_x} \\ \frac{v-c_y}{f_y} \\ 1 \end{bmatrix} z$$

Siendo los parámetros los mismos que en el caso anterior.

Transformaciones

Por último será necesario realizar transformaciones, es decir, como se ve un punto del espacio, desde otro punto de vista.

En este momento entra en juego el concepto de matriz de transformación, que tiene la siguiente forma.

$$T_{ab} = \begin{pmatrix} R_{3x3} & T_{3x1} \\ 0_{1x3} & 1 \end{pmatrix}_{4x4}$$

Siendo a y b dos referencias distintas, al multiplicar esta matriz por un punto cuya referencia es b, devuelve como se vería el punto desde la referencia a, es decir:

$$^{a}p = T_{ab}\ ^{b}p$$

En este caso en particular la transformación más común que se va a realizar es, de la referencia de la cámara a la del cubo (T_{gc}), y su inversa, del cubo a la cámara (T_{gc}). Se calculan ahora las operaciones que hay que seguir para pasar un punto de referencia cámara (c) a cubo (g), y viceversa. Para ello se parte de que es conocida T_{gc}.

$$^{g}p = T_{gc}\ ^{c}p = \begin{pmatrix} R_{gc} & t_{gc} \\ 0_{1x3} & 1 \end{pmatrix} \begin{pmatrix} c_x \\ c_y \\ c_z \\ 1 \end{pmatrix} = R_{gc}\,^{c}p + t_{gc}$$

Cabe destacar que el punto ^{c}p, tiene cuatro componentes, esta cuarta componente se trata de un factor de escala que no altera para nada el resultado, pero que si no estuviera no se podría llevar acabo la operación ya que la dimension de la matriz y la del vector no coincidirían. Se pasa ahora a calcular la transformación del cubo a la cámara.

$$T_{cg} = T_{gc}^{-1} = \begin{pmatrix} R_{cg} & t_{cg} \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} R_{gc}^{T} & -R_{gc}^{T}t_{gc} \\ 0 & 1 \end{pmatrix}$$

Por lo que

$$^{c}p = T_{cg}\ ^{g}p = \begin{pmatrix} R_{gc}^{T} & -R_{gc}^{T}t_{gc} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} g_x \\ g_y \\ g_z \\ 1 \end{pmatrix} = R_{gc}^{T}(^{g}p - t_{gc})$$

Creación del cubo

A continuación se describen las operaciones para lograr que todos los mapas partan del mismo punto.

En primer lugar se define la estructura soporte para toda la información, se trata de un cubo, en el que se va a colocar toda la información para leerla desde el punto de vista que se desea.

Primero se elige una posición de cámara que será de donde partirán todos los mapas al finalizar el método. En este caso se elige la posición de la cámara del primero de los mapas introducidos, por simplificar.

A partir de él se construirá el cubo, por lo que se buscan las distancias mínima y máxima dentro del depth map, estos serán los límites del cubo. Esta elección tiene sus desventajas ya que una mala elección del la referencia final de los mapas podría hacer que se perdiese información y que el resultado final fuera peor.

Para construir el cubo en sí, una vez conocidas la distancia mínima y máxima se proyecta a través de los vértices superior izquierdo e inferior derecho del mapa, la máxima distancia en el espacio, esos serán los extremos del fondo del cubo, una vez hecho eso sabiendo la profundidad que ha de tener se puede determinar la posición de los ocho vértices del cubo.

A la hora de realizar operaciones en el obviamente habrá que discretizarlo, para ello se define una resolución para cada una de las tres dimensiones, este valor indicará en cuantas partes se va a dividir por cada eje, por lo que el resultado será un cubo lleno de cubos más pequeños en su interior que se denominan vóxeles, el número de estos cubos será $resolucion_x \cdot resolucion_y \cdot resolucion_z$.

Ahora para cada mapa de profundidad que se quiere virtualizar se realizan las siguientes operaciones.

Escritura en el cubo

Una vez conocidos los parámetros del cubo, el primer paso es rellenarlo con los datos, para ello el primer paso es localizar la cámara en cuestión respecto al cubo. Es decir hay que calcular T_{gc}. Para ello son conocidos los siguientes datos:

- $T_{gc_{ini}}$ Siendo c_{ini} la cámara referencia, sobre la que se ha construido el cubo.
- $T_{wc_{1..n}}$ Siendo w un punto en el espacio desde el que están localizadas todas las cámaras.

Por lo que para conseguir T_{gc_n} solo habrá que multiplicar las matrices de transformación correspondientes, esto es.

$$Tc_{ini}c_n = (T_w c_{ini})^{-1}\ T_{wc_n}$$

$$T_{gc_n} = T_g c_{ini}\ Tc_{ini}c_n$$

Una vez hecho esto el siguiente paso es proyectar al depth map cada uno de los vóxeles del cubo, para así consultar su valor. Para ello se van a seguir los siguientes pasos para cada voxel.

- En este punto son conocidos los índices del voxel, por lo que habrá que calcular las coordenadas reales del centro del mismo, algo sencillo conociendo las dimensiones del cubo y la resolución de cada eje.
- Una vez conocidas las coordenadas habrá que pasarlas al sistema de referencia de la cámara. Para ello habrá que multiplicar el punto respecto del cubo $({}^g p)$, por la matriz de transformación T_{cg}, o lo que es lo mismo $(T_{gc})^{-1}$ como se ha visto en el apartado de operaciones.

- Una vez localizado el punto en referencia de la cámara hay que proyectarlo al mapa de profundidad para conocer su valor. Para ello se realiza la operación oportuna, definida en el apartado de operaciones.

  Una vez hecho esto hay que asegurarse de que el punto caiga dentro del mapa, ya que puede que ese hipotético punto de la escena no sea visible desde el punto de vista de la cámara en la que se esta trabajando. Si no es visible se dejará a 0 el valor de ese voxel, en caso de que la proyección este dentro de los límites del mapa se ha de calcular el valor a escribir en ese voxel.

  Este cálculo es la resta de la distancia real al objeto de la escena, es decir, la almacenada en el depth map, y la distancia en z de la cámara al centro del voxel, es decir, la componente z del punto en el espacio visto desde la referencia de la cámara que se ha calculado anteriormente. Este valor no se escribe directamente en el voxel, aunque sería una opción, antes de escribirlo, se aplica una función *tsdf (truncated signed distance function)*, esta función es de la siguiente forma.

$$f(d) = \begin{cases} \delta & d \geq \delta \\ 0 & d \leq -\delta \\ d & -\delta < d < \delta \end{cases}$$

  Siendo δ un valor elegido a la hora de generar los parámetros, y d el resultado del cálculo descrito anteriormente. Con esto se consigue que en el cubo se pueda intuir una superficie en el momento que se encuentre un cambio de signo. La razón de aplicar esta función *tsdf* es para acelerar la lectura del cubo, que será el siguiente paso.

E.1.1. Lectura del cubo

Por último se ha de realizar la lectura de los datos que después del apartado anterior han quedado en el cubo. Para ello, para cada pixel del mapa de profundidad a generar se realizan los siguientes cálculos.

- En primer lugar se calcula la dirección normalizada del origen de la cámara deseada al pixel mediante *Back proyection*, tras esto se transforma este vector, asi como el origen de la cámara a coordenadas del cubo para facilitar las operaciones posteriores.

- A continuación se calcula el punto de inicio y el punto final a muestrear, es decir,los límites del rayo proyectado que coinciden con el cubo.

- Una vez hecho esto se realiza el muestreo, comenzando por la primera intersección del cubo con el rayo, y usando como paso la máxima distancia que asegura que se van a recorrer todos los vóxeles intersectados, esta es.

$$\text{mín}(\frac{d_x}{res_x}, \frac{d_y}{res_y}, \frac{d_z}{res_z}) * 0{,}5$$

  Siendo d la distancia de la arista del cubo en cada una de las dimensiones.

- Para cada una de las muestras se tiene un valor en el espacio, el siguiente paso será encontrar el voxel al que corresponde ese punto, se trata de una operación sencilla ya que la coordenada a consultar se encuentra respecto de sistema de referencia del cubo, por lo que sólo será necesario dividir estas coordenadas reales entre la resolución de cada eje del cubo.

- Una vez consultado el valor del voxel en cuestión se compara con el consultado en la muestra anterior, de modo que si se detecta que ha habido un cambio de signo, significará que se a atravesado la superficie.

- Tras esto aún se realiza una interpolación consultando el intervalo en el que se ha detectado la superficie para estimar mejor la distancia real.

- Una vez conseguida esta distancia se ha de trasladar de nuevo a la coordenada origen. Hay que recordar que en el mapa de profundidad se almacenan las distancias en el eje z, y no en línea recta desde el origen que es el dato que se conoce en este momento, por lo que habrá que calcular esta distancia.

  El valor obtenido de este cálculo será el introducido en el mapa de profundidad trasladado ya a la referencia deseada.

Una vez realizados todos los pasos anteriores para cada uno de los mapas de entrada ya están dispuesto para realizar la fusión.

E.1.2. Programación

A continuación se enumeran las clases utilizadas para realizar el proceso anterior, en cuanto a su programación interna es seguir los pasos descritos anteriormente por lo que no se va a hacer hincapié en ella.

- *ReadDepthInfo* : Se encarga de la lectura de mapas de profundidad de un directorio determinado así como de la información de las cámaras desde las que están construidos estos detph maps.

  Para lograr esto se requiere que los mapas y los parámetros de las cámaras tengan un formato concreto.

- *RangeImageOp* Se trata de una clase con operaciones muy útiles con mapas de profundidad como las siguientes:

  - Normalizar y desnormalizar mapas de profundidad, ya que los mapas para realizar operaciones con ellos como añadir ruido o realizar la fusión necesitan no estar normalizados, sin embargo para representarlos si que hay que normalizarlos, de ahí la gran utilidad de este tipo de funciones.
  - Añadir ruido, esta es una utilidad también muy útil de esta clase, con ella se puede introducir en la imagen de forma sencilla ruido gausiano, o datos espurios en un porcentaje de la imagen que se requiera.
  - Por último permite el calculo del índice de señal ruido de un mapa de profundidad.

  En resumen es una clase muy útil para este proceso, tanto como lo será para la fusión.

- *MatrixTransf* Se trata de una clase que define una matriz de 3x4, así como ciertas operaciones sobre ella útiles para cálculos con matrices de transformación.

- *CudaCube* Es la más importante de las anteriores es la que se encarga de realizar el proceso anterior propiamente dicho. Este se es totalmente paralelizable por lo que el corazón del mismo esta programado en *CUDA*

  Por ello esta clase en realidad está compuesta de dos una en *C++* y otra en *CUDA*, en la parte de *C++* simplemente realiza las llamadas oportunas a la clase en *CUDA* del mismo nombre que será la que realizará los cálculos en paralelo.

  Las principales funciones de esta clase son las siguientes:

  - *setDimFromDepthMap*: Esta es la única de las tres funciones que se van a describir que se ejecuta enteramente en la cpu, se encarga simplemente de calcular los parámetros del cubo, este cálculo lo realiza usando la cámara del primer depth map que se ha leído.
  - *fillCube*: Como su propio nombre indica se encarga de rellenar el cubo que se ha generado mediante la función anterior, con los datos de un depth map cualquiera.

    Esta función esta paralelizada para que se ejecute una vez por cada pixel de la cara XY, y dentro de cada kernel se recorra toda la profundidad del cubo.

- *getVirtualDepthMap*: Realiza el último paso del proceso, es decir lee los datos del cubo desde la posición deseada para conseguir el depth map virtual.

  Esta función realiza el mismo proceso una vez para cada pixel del nuevo depth map, por lo que la paralelización es evidente.