Universidad de Valladolid

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

Grado en Ingeniería de Tecnologías de Telecomunicación

Implementación de una herramienta de visualización de algoritmos iterativos desarrollados en OpenCL

Alumno: Mario Calle Martín

Tutor/a/es: Javier Royuela del Val
Federico Simmross Wattenberg

A mis padres

Agradecimientos

En primer lugar, agradecer al Laboratorio de Procesado de Imagen, y de una forma especial a Javier y Federico, mis tutores en este trabajo, por su ayuda, paciencia y comprensión a lo largo de todo este tiempo.

De igual manera, agradecer a mis amigos, que han sido siempre amables, y han estado constantemente dispuestos a echar una mano.

Por último, mi eterno agradecimiento a mis familiares, en particular, a mis padres por sus incesantes ánimos y por estar siempre a mi lado.

Resumen

Con este Trabajo Fin de Grado se persigue simplificar las tareas de visualización de datos multidimensionales pertenecientes a un algoritmo iterativo. Este algoritmo puede estar implementado en una GPU, lo cuál complica el proceso de visualización de datos. Para solucionar este problema, se ha desarrollado una interfaz gráfica cómoda y sencilla que ofrece herramientas de adaptación de un algoritmo a la interfaz, diferentes formas de representación de los datos, modificación dinámica de los parámetros y datos de entrada, y aporta una mayor distinción en las etapas de ejecución de un algoritmo.

Índice general

Índice de figuras

Capítulo 1

Introducción

En este capítulo se presenta la problemática que motiva este trabajo. Las fases que se han llevado a cabo para finalizar el trabajo y los medios empleados.

1.1. Motivación

En la actualidad, en el campo de la ciencia e investigación científica, el uso de algoritmos para resolver problemas es un caso común. Estos algoritmos llegan a ser, en ocasiones, muy complejos, de los cuales, muchos de ellos tienen un carácter iterativo. Este carácter iterativo hace referencia a que la solución depende de las soluciones obtenidas en las diferentes instancias del problema [1].

Hoy en día, existen multitud de entornos de desarrollo, en alguno de ellos resulta complicado, e incluso tedioso, ver lo que está ocurriendo mientras se ejecuta el algoritmo deseado. Lo que dificulta el seguimiento de la evolución del algoritmo y la depuración de posibles errores.

Para solucionar este inconveniente, se podría exportar los datos de interés, y comprobarlos de la forma más apropiada para cada caso, por ejemplo, visualizándolos como imagen. Pero también puede llegar a ser un problema, ya que, en el enfoque de algoritmos iterativos interesa comprobar los datos entre iteraciones, al menos de una forma simple, para corroborar el buen funcionamiento del algoritmo.

Otra solución para la visualización de datos, son las interfaces gráficas. Que permiten añadir funcionalidades a los entornos de desarrollo. En este caso, facilitando, al usuario, el trabajo de exportar manualmente los datos y su posterior comprobación de la iteración. Es decir, las interfaces gráficas nos pueden servir de ayuda dándonos retroalimentación, por ejemplo, en forma de imágenes o gráficas. Lo cuál, nos permite visualizar datos complicados, o de alta dimensionalidad.

Concretando un poco más, una parte de estos algoritmos están pensados para una ejecución de tareas en paralelo, de forma que optimicen el tiempo necesario para conseguir su objetivo. Por ello, están implementados para que se ejecuten en una GPU (Graphics Processing Unit), donde se agravan los problemas que comentábamos antes, es decir, la forma de depuración de los datos y de su visualización, en cada iteración, debido a que

los datos se encuentran típicamente en la memoria de la GPU y su visualización implica pasos adicionales.

Parece apropiado entonces intentar adaptar este tipo de algoritmos a una interfaz gráfica, que permita la facilidad de uso y comprobación del algoritmo.

1.2. Objetivos

El objetivo principal que se trata de cumplir en este trabajo, es el desarrollo de una herramienta gráfica que permita visualizar datos multidimensionales a lo largo de la evolución de un algoritmo iterativo.

Como subobjetivos adicionales, se implementan los siguientes:

- Interfaz modular, es decir que para poder incluir un algoritmo en la interfaz, no sea necesario modificar el código fuente. Simplemente será requerido registrar el algoritmo en forma de un módulo independiente.
- Soporte de representación de datos alojados en la GPU, sin necesidad de lectura previa por parte del usuario en la CPU (Central Processing Unit), en cada iteración.
- Soporte de representación de datos complejos.
- Carga de datos y edición de parámetros dinámica, para cada algoritmo.
- Posibilidad de elegir la representación de datos, en forma de imagen o de gráfica (Un plot).

1.3. Fases

Las fases que se han seguido para alcanzar los objetivos anteriormente citados son las siguientes:

- Documentación de las tecnologías que serán empleadas para dar funcionalidad a la interfaz. Entre estas tecnologías se incluyen, OpenCL, para el procesamiento de datos en la GPU; Qt, para la implementación de la interfaz y sus funcionalidades; OpenCV, para la representación de datos a través del uso de matrices.
- Planteamiento de los requisitos, funcionales y no funcionales, de la interfaz de usuario [2]. Y puntualización del conjunto de funcionalidades específicas a desarrollar.
- Diseño preliminar de la interfaz y diagrama de clases.
- Diseño de la interfaz, seguido de su implementación. Sucesivamente, se sugieren nuevas funcionalidades y se optimiza el rendimiento de la interfaz.
- Implementación de algoritmos de prueba.
- Una vez que se ha comprobado el funcionamiento de la interfaz con un ejemplo simple, que pone a prueba cada uno de los requerimientos, se verifica la correcta puesta en marcha de la interfaz, con la adaptación a ella, de un algoritmo más complicado.

1.4. Medios utilizados

Para la realización de este trabajo se han usado varios lenguajes de programación, tales como C++ y OpenCL, bibliotecas necesarias y que amplían la funcionalidad de los lenguajes de programación, son el caso de Qt y OpenCV. Adicionalmente, se han empleado varias herramientas y equipos.

Dentro de las herramientas software, se ha utilizado:

- MatLab como software de cálculo numérico, donde se han hecho pruebas, para después implementar correctamente las funciones de la interfaz.
- KDevelop, software usado para hacer pruebas en relación a los lenguajes de programación.
- QtCreator, software empleado para diseñar la interfaz, y toda su lógica interna.

En cuanto a equipos hardware, se ha dispuesto de un ordenador de sobremesa con distribución Linux, con las siguientes características:

- Procesador Intel(R) Core(TM) i7-4790 CPU @ 3.60 GHz
- Memoria (RAM) 16.0 GB
- Almacenamiento interno 256 GB HDD
- Tarjeta Gráfica NVIDIA GEFORCE GTX 970

La versión usada de Qt fue la 5.9.2.
La versión usada de OpenCV fue la 3.3.1.
La versión usada de OpenCL fue la 1.2.

Finalmente, los algoritmos desarrollados por el departamento LPI, para poder probar la interfaz:

- Resolución de sistemas de ecuaciones lineales mediante gradiente conjugado. Aplicación a recuperación de imagen de RM compleja a partir de proyecciones arbitrarias
- Algoritmo de registrado grupal no rígido. Aplicación a imagen cardíaca de RM 2D+t.

Capítulo 2

Tecnologías empleadas

En este capítulo se entrará más en detalle en las herramientas específicas que se han utilizado durante el desarrollo de este trabajo.

2.1. Interfaces Gráficas de Usuario

2.1.1. Introducción a las Interfaces Gráficas de Usuario

Una interfaz de usuario es la parte de un ordenador y su software, que una persona puede ver, oír, tocar, y hablar. O dicho de otra forma, entender y dirigir. La interfaz de usuario básicamente está formada por dos componentes, la entrada y la salida. La entrada o input, es la forma de comunicar las necesidades de una persona al ordenador, por ejemplo a través del uso de un teclado, un ratón, o incluso la voz. La salida o output, es como transmite el ordenador los resultados que el usuario ha requerido, actualmente los mecanismos más populares que realizan esta función son las pantallas, apoyados por la emisión de voz y sonido.

Una interfaz diseñada correctamente, prestará una combinación adecuada de mecanismos de entrada y salida que satisfagan las necesidades, capacidades y limitaciones del usuario, de la forma más efectiva [3].

En una interfaz gráfica de usuario, también conocida como GUI (del inglés Graphical User Interface), el principal mecanismo de interacción es un puntero, que asemeja el comportamiento de la mano del hombre. Al conjunto de elementos, con los que el usuario interactuará, se les denominados objetos. Los cuales pueden ser vistos, oídos o tocados, es decir, percibidos por el usuario. Estos objetos son visibles al usuarios y se utilizan para realizar tareas, denominadas acciones. Las tareas que el usuario desea realizar, se llevarán a cabo accediendo y modificando los objetos mediante su propia selección, modificación o indicación, teniendo cada objeto un comportamiento resultante estándar.

Ventajas y Desventajas de los Sistemas Gráficos

A continuación se listarán las ventajas que presentan los sistemas gráficos frente a los basados en texto.

- Los símbolos se reconocen más rápido y con mayor precisión que el texto, ya que los atributos gráficos de los iconos como el color o la forma son de gran utilidad para clasificar los elementos.
- Un aprendizaje más rápido y fácil, debido a la capacidad de retener con sencillez las imágenes y símbolos.
- La representación, visual o espacial, de la información facilita su propia retención y manipulación, lo que conduce a una consecución de la resolución del problema.
- Da al usuario una mayor sensación de control. Esto se debe a que el usuario da órdenes con facilidad, e incluso puede revertir esa orden o los cambios realizados.
- El aporte de retroalimentación es inmediato, por lo que las acciones realizadas por el usuarios se pueden observar con presteza.
- Aportan un mayor atractivo, ya que son más entretenidos y simples de utilizar. Simpleza en cuanto a la no necesidad de escritura constante puesto que se dispone de un puntero de selección con lo que se controla el sistema.
- Pueden ocupar menos espacio, debido a la sustitución de texto por un símbolo. Es posible dar más información con un elemento gráfico de cierto tamaño que el texto necesario para explicarlo. Aunque no siempre se cumple este caso.
- Los elementos gráficos llegan a ser universales, por lo que el sistema se globaliza un poco más.

Como desventajas se citarán las siguientes.

- Complejidad de un buen diseño. La cantidad de herramientas disponibles para el diseño gráfico de interfaces son muy numerosas respecto a las basadas en texto, lo que puede derivar en un mal diseño si no se deciden adecuadamente las elecciones.
- Siempre hace falta un mínimo de aprendizaje. La primera vez que vemos una interfaz gráfica, no todo está tan claro de inmediato. El significado de algunas palabras o iconos pueden ser desconocidos para el usuario. Por lo que llevará un tiempo de adaptación al nuevo sistema.
- Las interfaces requieren un uso de ventanas gráficas donde poder mostrar el contenido. Este empleo y manipulación de ventanas puede llegar a consumir mucho tiempo, y hacerse repetitivo.

- El uso de una interfaz gráfica puede incrementar la confusión del usuario, puesto que no se garantiza un orden de elementos en la pantalla para cada usuario. Para dejarlo más claro, el orden que haya impuesto el diseñador a su interfaz puede desorientar a una suma de potenciales usuarios.

- Limitaciones de hardware. Una buena interfaz gráfica siempre va a requerir más recursos, en cuanto velocidad de procesamiento, resolución de pantalla y capacidad gráfica, que una interfaz basada en texto. Si no se dispone de un hardware potente que soporte la interfaz a usar, se acotará el potencial de dicha interfaz.

Características de las Interfaces Gráficas de Usuario

El conjunto de conceptos que posee una interfaz gráfica de usuario se puede resumir en los siguientes puntos:

- Presentación visual sofisticada: hace referencia al aspecto visual de la interfaz, lo que el usuario ve en la pantalla. La sofistificación de un sistema gráfico permite representar líneas, iconos e imágenes. Adicionalmente, posibilita el uso de una amplia variedad de fuentes de caracteres, diferenciando entre tamaños y estilos. También se puede realizar una interfaz más compleja o avanzada mediante el uso de animaciones de fotogramas y vídeos. Los principales elementos de la interfaz que se presentan visualmente incluyen ventanas, barras de menú, iconos, controles (cajas de texto, botones, listas de selección, barras de desplazamiento...), y un cursor o puntero.

- Interacción: los elementos de la pantalla gráfica sobre los que se ejecutará alguna acción deben ser inicialmente identificados. Es decir, el usuario debe seleccionar primero el elemento para una acción en concreto y seguidamente hacer un click o pulsación sobre ese elemento para enviar una señal al sistema. El mecanismo principal que cumple esta función suele ser el puntero o cursor sobre los botones de la interfaz. El teclado queda como mecanismo secundario que puede llegar a cumplir esa función.

- Conjunto de restricciones de las opciones de la interfaz. El grupo de alternativas disponibles al usuario es lo que se encuentra representado en la interfaz. O los posibles resultados que se pueden obtener a través de la interfaz está indicado en la pantalla. Este concepto se conoce como WYSIWYG (del inglés What You See Is What You Get), que nos da a entender, que lo que ves es lo que obtienes.

- Visualización: concepto que hace referencia a la forma mas eficaz de representar la información y dando prioridad a la información más relevante. Creando una estructura que permita al usuario entender todo lo que perciba.

2.1.2. Qt

Qt es una biblioteca multiplataforma, para la creación de entornos gráficos, desarrollada como un software libre y de código abierto por la compañía The Qt Company [4]. Esta biblioteca está programada con el lenguaje C++.

Qt posee una herramienta denominada QtCreator, que permite diseñar e implementar la interfaz gráfica con mayor facilidad, ya que es un entorno de desarrollo integrado (IDE), que ofrece la capacidad de completar y resaltar código para una mejor organización, un sistema de ayuda, una herramienta de depuración de código y un control de versiones.

Parte Gráfica de Qt

Para la creación de los elementos gráficos del entorno, QtCreator dispone de una amplia utilidad, sencilla de usar, la que permite añadir, eliminar y modificar los diferentes componentes que constituyen la interfaz gráfica.

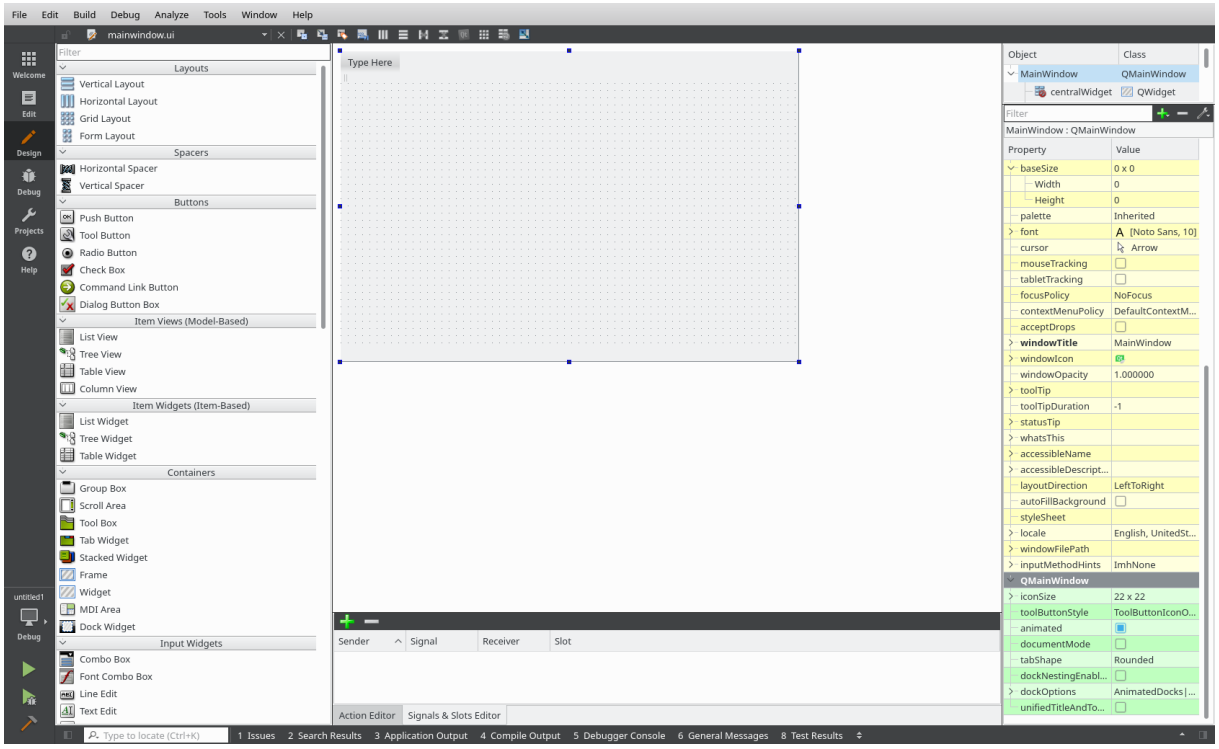

Figura 2.1: Utilidad de QtCreator para el diseño del entorno

En la figura 2.1, se muestra la pantalla de diseño para un proyecto nuevo. Principalmente son tres paneles importantes, el panel izquierdo contiene todos los elementos básicos, el panel central superior es el espacio donde se colocarán todos los objetos, y el panel derecho inferior que va mostrando los atributos del objeto seleccionado lo que supone una gran ayuda y comodidad a la hora de modificar dichos atributos.

El diseño en Qt está basado en el uso de layouts, widgets, signals y slots. Un layout es un elemento que se emplea para organizar los widgets, pudiendo adaptar el conjunto de la interfaz a diferentes resoluciones. Widgets son los elementos básicos para crear interfaces (botones, listas de selección, cuadros de texto ...). Signals son señales que se emiten cuando ocurre un evento particular. Slots son las funciones o métodos que se ejecutan como consecuencia de la emisión de una determinada señal.

Los elementos en Qt, al ser una programación orientada a objetos, poseen atributos y métodos heredados de otras clases, además de los suyos propios. Por ejemplo en la figura 2.2 se pueden observar los diferentes atributos, los propios y heredados, del objeto pushButton.

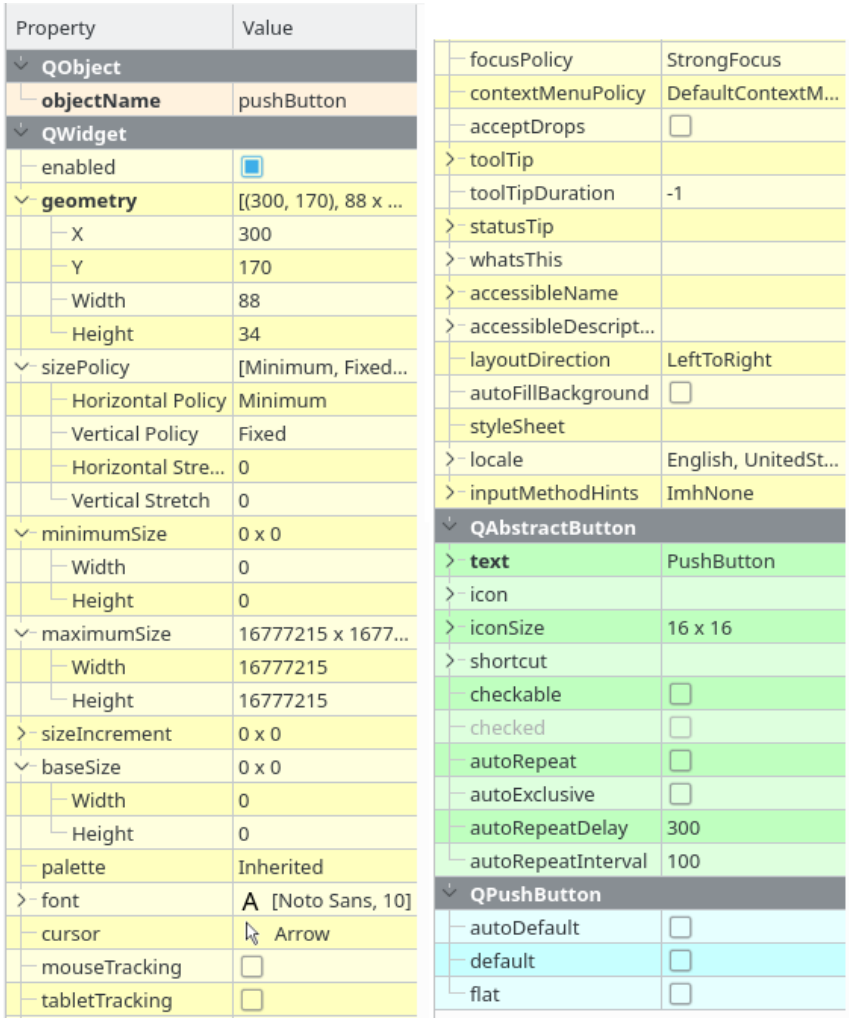

Figura 2.2: Atributos de la clase PushButton

Parte Lógica de Qt

Una vez elaborado el apartado gráfico es necesario implementar el comportamiento de todos los elementos empleados. Para ello se usan las señales (SIGNALS) que representan las diferentes acciones que un elemento de la interfaz puede desencadenar. Cada señal se puede emparejar con una ranura,de espacio (SLOT), donde se encontrará implementado el comportamiento que desarrollará la interfaz. Para facilitar la identificación de las señales de cada elemento, QtCreator nos ayuda mostrando todas las disponibles para cada uno. En la figura 2.3 se reflejan todas las señales posibles para un objeto de la clase PushButton, correspondiente a un botón.

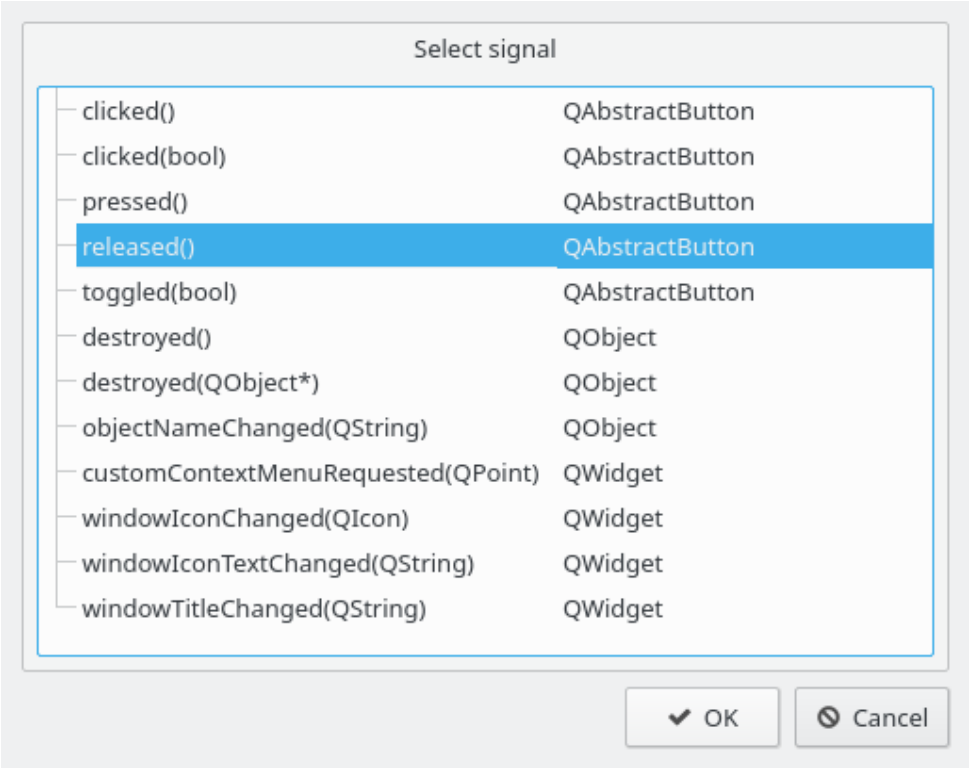

Figura 2.3: Señales de la clase PushButton

Para aclarar un poco más el funcionamiento de las señales, tomando como ejemplo la cuarta opción de la figura 2.3, la señal *released()* hace referencia al momento cuando es pulsado el botón correspondiente y se suelta. Suponiendo que se ha implementado un comportamiento en la ranura relacionada con dicha señal, se llevará a cabo esa función.

Es muy probable que en alguna ocasión se necesite obtener información o modificar algún atributo de un elemento para cambiar el comportamiento de una parte de la interfaz. Esto es posible gracias a los típicos métodos get y set particulares de cada clase. Tomando como ejemplo un *checkbox* o casilla seleccionable, el método que permite modificar el estado de selección es *setChecked(bool)* y en este caso no es un get sino un is, para consultar el estado en el que se encuentra, *isChecked()* (nos devolverá verdadero o falso).

Con esta modificación dinámica de los objetos se consigue controlar la apariencia y la conducta de la interfaz. Por ejemplo, se previene el uso de ciertas partes de la interfaz hasta el momento que sean necesarias para evitar posibles incompatibilidades, y visualización de las zonas deseadas y capacidad de ocultar el resto.

2.2. Computación en GPU

2.2.1. Introducción a la Computación en GPU

La computación en GPU o GPGPU (General-purpose computing on graphics processing units) hace alusión al beneficio que supone el uso de una GPU para realizar cálculos, en aplicaciones de ingeniería, medios digitales y científicos, comparado con una unidad de procesamiento central o CPU. La GPU actúa como un suplemento del procesador principal en el cuál se pueden acelerar las aplicaciones gracias a su enorme potencia de procesamiento paralelo en comparación con el diseño de núcleo múltiple de las CPU.

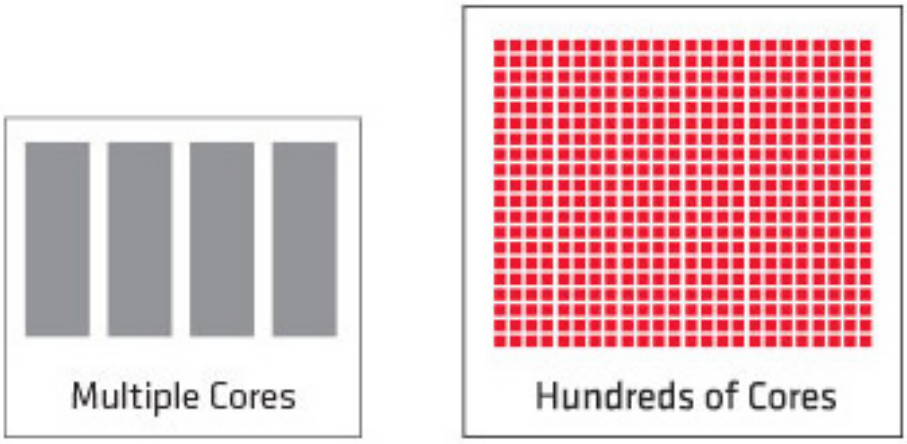

Figura 2.4: Comparación de núcleos en una CPU izquierda, y una GPU derecha.[5]

Este concepto de computación en GPU comienza a hacerse popular a partir del año 2001, gracias a la innovación que supuso el soporte de sombras programables y la representación de números racionales muy grandes o muy pequeños de una forma eficiente en los procesadores gráficos. Dando lugar a diversos experimentos en el mundo científico, como una rutina de multiplicación de matrices y posteriormente a demostraciones en las que se exponía la velocidad de procesamiento, en paralelo, de una GPU comparada con una CPU usando una implementación de la factorización LU (descomposición de una matriz en un producto de dos matrices, una triangular inferior y una superior)[6].

Las arquitecturas de procesadores modernas han acogido el paralelismo como una vía de mejora del rendimiento. En el caso de las CPUs añadiendo varios núcleos, o en el caso

de las GPUs añadiendo la capacidad de programar sus procesadores en vez de ser sólo dispositivos de computación de gráficos. Hoy en día los sistemas informáticos incluyen CPUs con avanzada capacidad de procesamiento paralelo, GPUs y otros tipos de procesadores, por lo que es importante disponer de un software que los programadores puedan utilizar para obtener beneficio de esta heterogeneidad de plataformas de procesamiento [7].

Actualmente existen dos grandes lenguajes de programación dedicados a la computación en GPU con propósito general. Son el caso de CUDA y OpenCL. CUDA, creado por NVIDIA, ha sido especialmente diseñado y optimizado para las tarjetas gráficas de dicha marca por lo que no funciona fuera de ese conjunto [8]. OpenCL es un estándar abierto desarrollado por el Grupo Khronos [9], que soporta una amplía variedad de plataformas.

2.2.2. OpenCL

Como se ha indicado anteriormente, OpenCL es un estándar para la programación de diversos conjuntos de CPUs, GPUs y demás dispositivos dedicados a la computación, organizados en una plataforma. Es un entorno de trabajo o framework, para la programación en paralelo que incluye un lenguaje, una API (Application Programming Interface), una biblioteca y un sistema de rutinas para el desarrollo de software. Usando OpenCL, es posible escribir programas de propósito general que funcionen en la GPU sin necesidad de adaptar los algoritmos a una API gráfica 3D como OpenGL o DirectX [7].

Para describir las ideas básicas de OpenCL se emplea una jerarquía de modelos [10]:

- Modelo de Plataforma
- Modelo de Memoria
- Modelo de Ejecución
- Modelo de Programación

Modelo de Plataforma

En este modelo se define una representación de alto nivel de cualquier plataforma heterogénea que esté usando OpenCL. Una plataforma OpenCL siempre incluye un host, que será el encargado de interactuar con el entorno externo al programa OpenCL, incluyendo las entradas y salidas o la interacción con el programa de usuario. El host está conectado a uno o más dispositivos. Estos dispositivos pueden ser una CPU, o GPU, u otro procesador soportado, y es donde se ejecutará la secuencia de instrucciones. A su vez los dispositivos OpenCL se dividen en unidades de cómputo y cada uno de ellos en uno o más elementos de procesamiento. La computación en un dispositivo se lleva a cabo dentro de los elementos de procesamiento (PEs). El modelo de plataforma se ilustra en la figura 2.5.

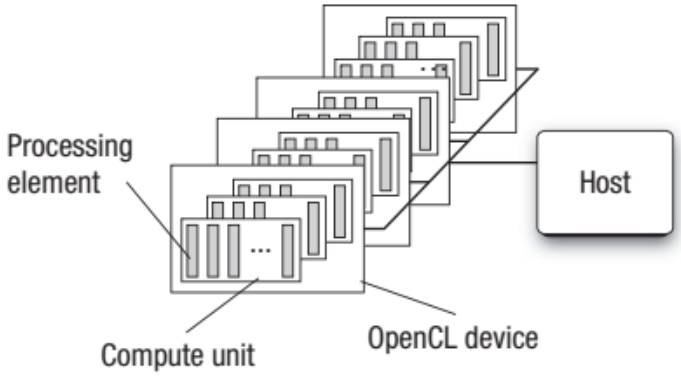

Figura 2.5: Plataforma OpenCL con un host y más de un dispositivo

Modelo de Ejecución

Una aplicación OpenCL está formada por dos partes, el programa que se ejecuta en el host, y una colección de uno o más kernels que se ejecutan en los dispositivos OpenCL. Estos kernels generalmente son funciones que transforman un objeto de la memoria de entrada en un objeto para la memoria de salida.

Un kernel se define en el host, y el host se encargará de enviar el kernel a un dispositivo. Cuando el host emite dicho comando el sistema OpenCL crea un espacio de índices, de modo que cada instancia del kernel se ejecute en cada punto de este espacio de índices. A cada instancia de ejecución del kernel se le denomina work-item, el cuál está definido por unas coordenadas en el espacio de índices (global ID). El espacio total o global se puede dimensionar en work-groups formados por el mismo número de work-items,por lo que se añaden más identificadores, el group ID que describe a cada work-group y el local ID que identifica a cada work-item dentro del work-group.

La primera tarea por parte del host será definir un contexto para la aplicación OpenCL, donde se especificará el entorno donde se definirán y ejecutarán los kernel. Para ser más preciso se define el contexto en términos de los siguientes recursos:

- Dispositivos, es la colección de dispositivos OpenCL que usará el host.
- Kernels, son las funciones OpenCL que se ejecutarán en los dispositivos OpenCL.
- Objetos de programa, es el código del programa y ejecutables que implementan los kernels.
- Objetos de memoria, es el conjunto de objetos en memoria que son visibles para los dispositivos OpenCL y contienen valores que pueden ser usados por cada instancia de un kernel.

Una vez definido el contexto, para que el host pueda interactuar con los dispositivos OpenCL es necesario definir una cola de comandos o command-queue, para cada dispo-

sitivo. El host envía comandos a esta cola y esperarán en ella hasta que sean ejecutados en el dispositivo OpenCL correspondiente.

Modelo de Memoria

En el modelo de memoria de OpenCL se definen cinco regiones distintas:

- Memoria del Host, solamente visible para el host.
- Memoria Global, en ella se permiten accesos de lectura y escritura para todos los work-items de todos los work-groups.
- Memoria Constante, esta región de memoria permanece intacta durante la ejecución de un kernel. Es decir, que los work-items tiene acceso solamente de lectura.
- Memoria Local, región dedicada a un work-group, accesible por todos los work-items que conforman ese work-group.
- Memoria Privada, espacio memoria destinado para un work-item, y no visible para el resto de work-items.

En la figura 2.6, se puede ver como está relacionados las diferentes partes de un dispositivo con el modelo de memoria.

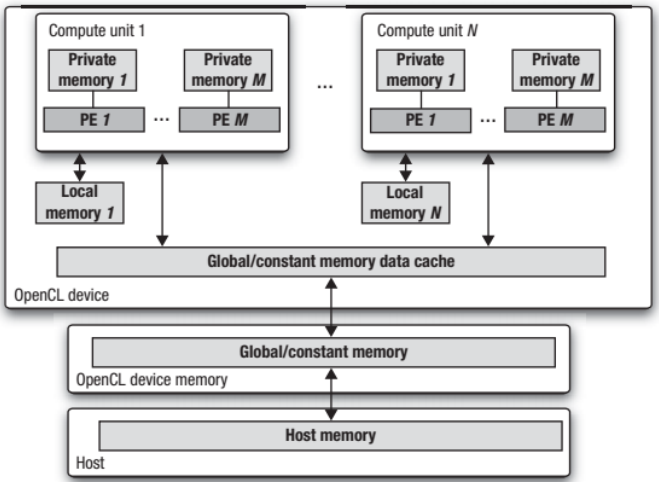

Figura 2.6: Modelo de memoria en relación con el modelo de plataforma

Modelo de programación

Para hacer más manejable el problema de la programación en paralelo se plantean dos modelos. Uno basado en el paralelismo de los datos y otro basado en el paralelismo de las tareas.

En la programación basada en el paralelismo de los datos se aplica una secuencia de instrucciones a los múltiples elementos de un objeto de memoria. El espacio de índices asociado con el modelo de ejecución define los work-items y como se mapea los datos en los work-items En la figura 2.7 se muestra un ejemplo.

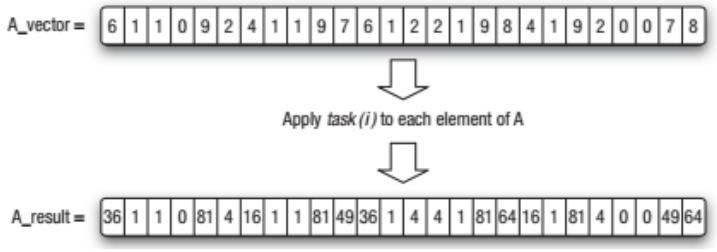

Figura 2.7: Ejemplo programación basada en el paralelismo de los datos. Se multiplica el elemento por si mismo, es decir la misma operación en cada elemento

En cambio, en la programación basada en el paralelismo de las tareas , el problema requiere de varias tareas y cada tarea ejecuta una única instancia del kernel independientemente del espacio de índices. El problema completo se resuelve cuando termina de ejecutarse la última tarea. Dependiendo del tiempo que requiera cada una, esa tarea acabará antes o después, aquí entra en juego el balanceo de carga, es decir, asignar un cierto número de tareas a cada elemento de procesamiento para aprovechar más el tiempo y aumentar el rendimiento.

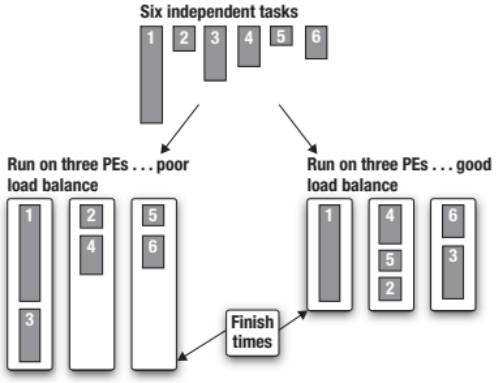

Figura 2.8: Ejemplo del balanceo de carga, distribuyendo las tareas y optimizando el tiempo.

Sumario OpenCL

Para concluir este apartado explicativo de OpenCL, se describen los pasos básicos y necesarios para conseguir poner en marcha una aplicación en una plataforma heterogénea.

- Descubrir los componentes que formarán el sistema heterogéneo

- Explorar las características de estos componentes para que el software pueda adaptarse a las particularidades de los diferentes elementos hardware.
- Crear los bloques de instrucciones (kernels) que se ejecutarán en la plataforma
- Preparar y manipular los objetos de memoria involucrados en la computación
- Ejecutar los kernels en el orden correcto y en los componentes del sistema que correspondan.
- Recoger los resultados finales.

2.2.3. OpenCV

OpenCV (Open Source Computer Vision Library) es una biblioteca de software libre, desarrollada inicialmente por Intel, diseñada para la eficiencia computacional y enfocada en la creación de aplicaciones de ejecución en tiempo real [11]. Está escrita en C/C++.

Esta biblioteca contiene mas de 2500 algoritmos optimizados, que pueden ser usados para la detección y el reconocimiento facial, la clasificación de las acciones humanas en los vídeos, el seguimiento del movimiento de cámaras u objetos, la extracción de modelos 3D de los objetos, representación de datos, y muchas más funciones [12].

Uno de los objetivos principales de OpenCV es ofrecer una infraestructura de visión artificial que sea fácil y simple de usar para ayudar al desarrollo de aplicaciones sofisticadas. Ya que la visión artificial y el aprendizaje automático están muy relacionadas, OpenCV también dispone de una biblioteca completa de propósito general para el aprendizaje automático [11], enfocada en el reconocimiento y asociación de patrones estadísticos.

¿Qué es la Visión Artificial?

La visión artificial o visión por computador, tiene un doble objetivo. Desde el punto de vista de la ciencia biológica, la visión artificial pretende conseguir modelos computacionales del sistema visual humano. Desde el punto de vista de la ingeniería, la visión artificial se enfoca en el desarrollo de sistemas autónomos que puedan llevar a cabo algunas de las tareas que desempeña el sistema visual humano. Muchas de estas tareas están relacionadas con la reconstrucción de información tridimensional, a partir del estudio de datos bidimensionales que varían en el tiempo (datos recogidos por una o varias cámaras) [13].

Debido a la complejidad del sistema visual humano y su capacidad de realizar muy bien algunas tareas, reconocimiento facial por ejemplo, la investigación en este campo es bastante difícil, por lo que los sistemas de visión artificial están limitados en comparación. Un ser humano puede reconocer rostros en muchos tipos distintos de situaciones, cambios de iluminación, puntos de vista diferentes, expresiones de los rostros, etcétera.

Suavizado de imágenes

Para mostrar la capacidad de OpenCV, se va a explicar varios ejemplos clásicos. El primero es el suavizado o emborronamiento. Es una técnica de procesado de imagen simple y de uso frecuente en la reducción de ruido de una imagen. Para lograr este efecto se aplica un filtro la imagen. El tipo más común de filtros es lineal, en el que el valor de un píxel de salida se determina como una suma ponderada de valores de píxeles de entrada. Dependiendo de los coeficientes del filtro (kernel) tendremos un tipo de filtro u otro. Siendo uno de los más populares, el filtro gaussiano. El funcionamiento básico de este filtro, es para cada píxel, dar un mayor peso al píxel central y menor peso a los adyacentes, cuanto más lejos un peso menor. Este ejemplo se implementa fácilmente con OpenCV [14] y en la figura 2.9 se ve el efecto de aplicar un filtro gaussiano a una imagen.

Figura 2.9: Aplicación OpenCV, Filtro Gaussiano.

Detección de bordes en imágenes

En el segundo ejemplo se propone la detección de bordes que existe en una imagen, mediante el uso de derivadas. Es decir cambios en el gradiente que indiquen un cambio brusco en la imagen. Para conseguir este objetivo, se emplea el operador Sobel. Este operador, calcula el gradiente de la intensidad de una imagen en cada píxel. Es decir que por cada punto se obtiene la magnitud del mayor cambio posible, la dirección de éste y el sentido desde oscuro a claro. El procedimiento es el siguiente, se calculan los cambios horizontales y verticales aplicando en cada uno un kernel impar, y en cada punto de la imagen se calcula la aproximación del gradiente, en ese punto, combinando los resultados de los cambios horizontales y verticales [15]. Este ejemplo se puede implementar con OpenCV, y en la figura 2.10 se muestra el efecto de aplicar el operador Sobel a la imagen para detectar los bordes.

Figura 2.10: Aplicación OpenCV, Operador Sobel.

Aplicación Concreta en este Trabajo

Para la realización de este trabajo OpenCV ha sido de gran ayuda gracias a la posibilidad que te ofrece OpenCV de utilizar matrices, en los lenguajes de programación C++ y OpenCL, como contenedores de datos (una imagen, por ejemplo) que facilitan la optimización de operaciones y funciones, y sobre todo la capacidad de incorporar estas matrices en algunas de las funciones que proporciona Qt.

El uso principal de este software ha sido la utilización de funciones que han permitido el procesamiento y la representación, en forma de imágenes, de los datos.

Capítulo 3

Proceso de Desarrollo

En este capítulo se detallarán los pasos seguidos para conseguir la consecución del objetivo principal planteado en este trabajo y de los subobjetivos.

3.1. Análisis

En esta fase del desarrollo, se han identificado los requisitos funcionales y no funcionales. Teniendo en cuenta que el objetivo principal es implementar una herramienta que permita la visualización de algoritmos iterativos desarrollados en OpenCL, se definen unos requisitos fundamentales.

- El sistema permitirá ejecutar un algoritmo cualquiera desde la interfaz.
- El sistema proporcionará representación de datos extraídos de la memoria del dispositivo OpenCL y de la memoria del host.
- El sistema aportará la visualización de los datos en forma de representaciones visuales o imágenes.
- El sistema permitirá al usuario la interacción con la interfaz después de cada iteración de su algoritmo.
- El sistema colocará y ordenará las diferentes visualizaciones requeridas por el usuario, en un espacio de la interfaz fácilmente accesible al usuario.

Una vez fijados estos requisitos funcionales, se plantean otros adicionales que aporten un mayor numero de utilidades al sistema.

- La aplicación soportará la representación de datos de varios algoritmos independientes al mismo tiempo.
- El sistema aceptará un cierta variedad de tipos de datos usados por el cliente.

- El sistema permitirá la edición dinámica de parámetros y datos de entrada de un algoritmo desde la interfaz, sin necesidad de modificar el código del algoritmo.
- La interfaz admitirá la reproducción de datos en forma de gráficas (plots).
- El usuario tendrá la posibilidad de exportar las representaciones deseadas y guardarlas.
- El sistema ofrecerá la capacidad de representar en varias disposiciones, diseños o layouts.
- El usuario poseerá el control sobre las iteraciones que desea realizar, y la forma en que se iterará. De una en una, un número concreto de veces seguidas, o hasta que el usuario considere oportuno (convergencia de un parámetro a un cierto valor, por ejemplo).
- El sistema presentará los diferentes algoritmos que están incorporados, y dará elección al usuario de cual desea elegir.

En cuanto a los requisitos no funcionales para este proyecto, se destacarán algunos de entre los muchos posibles.

- Rendimiento. El uso de la interfaz para trabajar con algoritmos debe afectar el mínimo posible al tiempo requerido por este algoritmo para completarse.
- Compatibilidad y portabilidad. Capacidad de la aplicación de ejecutarse e instalarse en distintos sistemas operativos. El cumplimiento de este requisito se cumple con facilidad, ya que el uso de la biblioteca Qt, al ser multiplataforma, funciona en varios sistemas operativos [4] (Linux, Windows, OS X, Android ...).
- Escalable. Se refiere a la adición de nuevas funcionalidades de una forma sencilla y práctica a la interfaz.
- Estabilidad y Robustez. El usuario debe poder realizar sus tareas sin necesidad de preocuparse por el código ajeno a su algoritmo. Y el programa tendrá que ser capaz de controlar los errores, producidos por el algoritmo empleado por el usuario, que afecten al funcionamiento del programa completo.
- Capacidad de prueba de la aplicación. Es la disposición que ofrece el programa para encontrar fallos (si existen) por el mero hecho de probar la aplicación.

En la figura 3.1 se muestra el diagrama de clases que da información acerca de los atributos y métodos de las clases y estructuras que se plantean. Además de la relación existente entre ellas.

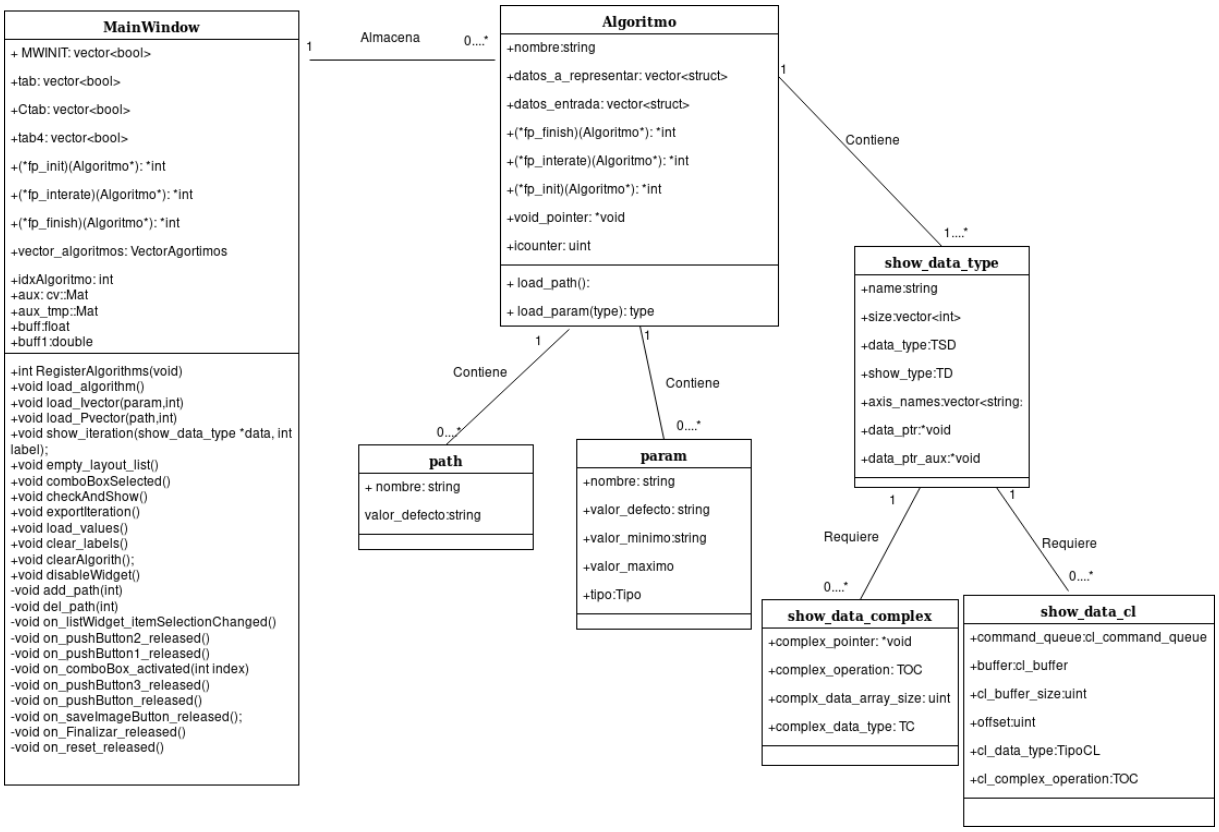

Figura 3.1: Diagrama de clases

3.2. Diseño

En este apartado de diseño se diferenciará entre diseño de la interfaz gráfica de usuario, y diseño de la aplicación. Se siguió este orden para conseguir una GUI cómoda y que resultara atractiva al usuario.

3.2.1. Diseño de la Interfaz Gráfica de Usuario

Se propone una estructura formada por varios paneles alineados horizontalmente, que cumplan con los requisitos correspondientes, citados en la sección anterior. Debido a que la aplicación será una interfaz de visualización de datos, parece lógico dar una prioridad al panel donde se representarán los datos, por lo que será el más grande y vistoso. Los demás paneles, siguiendo un orden natural de lectura (izquierda a derecha), se posicionan según el orden de uso.

Para explicar el posicionamiento de los paneles, primero se muestra una captura de la interfaz recién iniciada.

En la figura 3.1 se ha seleccionado un algoritmo que posee parámetros y datos de entrada para poder ilustrar ese panel.

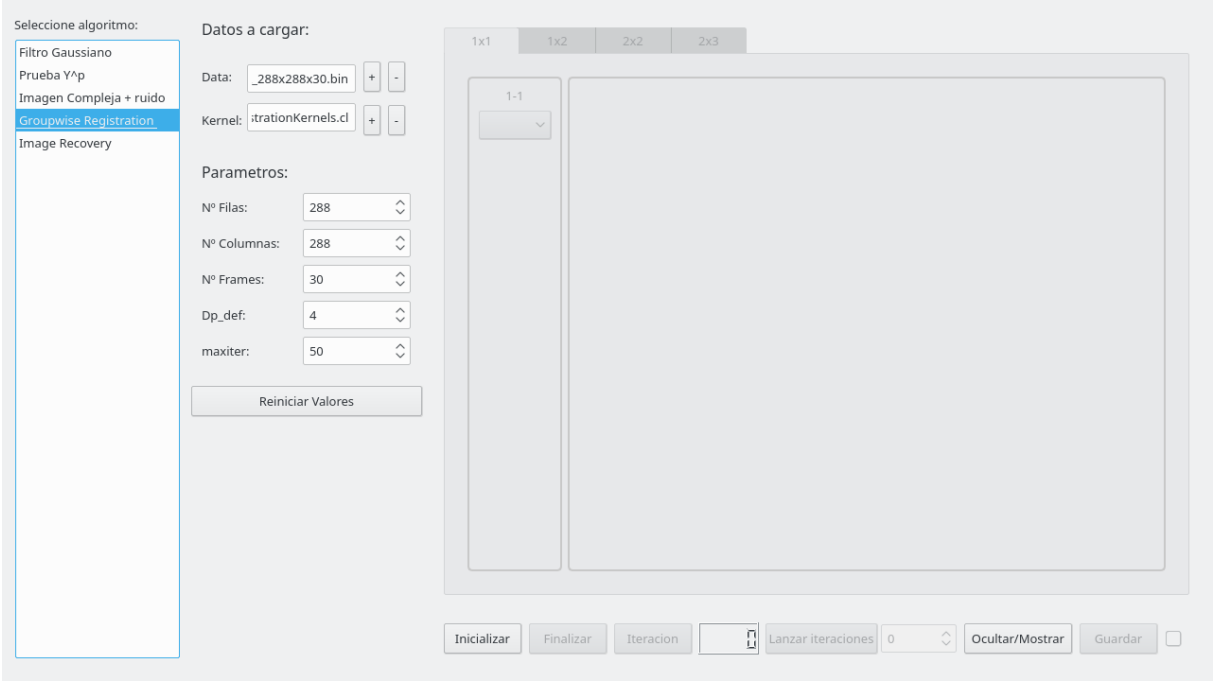

Figura 3.2: Interfaz gráfica diseñada

Como se puede observar en la figura anterior, la interfaz está dividida en tres partes. De izquierda a derecha, se encuentra en primer lugar el selector de algoritmos, espacio reservado para elegir el algoritmo con el que se va a empezar a trabajar. Se decidió colocar al inicio puesto que será el primer paso que dará el usuario. En el siguiente panel de la interfaz o panel central, se ve un espacio dedicado a mostrar los datos de entrada y parámetros necesarios por el algoritmo seleccionado. Este panel aporta un mayor dinamismo a la interfaz y al algoritmo que necesite modificar su valores iniciales en cada ejecución.

Finalmente, se ve un gran bloque a la derecha, que contiene un conjunto de botones en la parte inferior y un selector de pestañas en la parte superior. De nuevo, se mantiene el orden de uso en la botonera inferior, dando prioridad a los botones dedicados al control de ejecución del algoritmo y posicionando más a la derecha los botones usados en funciones adicionales. La parte más amplia de este gran panel derecho es una extensión dispuesta para la visualización de los datos. Este panel tiene la capacidad de incrementar su tamaño, obteniendo el espacio del bloque central, siempre y cuando se quiera ocultar dicho panel para dar una mayor relevancia a las representaciones.

3.2.2. Diseño de la Aplicación

Una vez definidos los diferentes componentes que constituirán la interfaz gráfica y fijado su diseño artístico se procede a explicar como se cumplirán todos los requisitos de la fase de análisis.

Antes de comenzar a usar la interfaz, el usuario bien si comienza desde cero o si tiene implementado el algoritmo, deberá adaptarse a un formato concreto compuesto por cuatro fases. Son todas ellas necesarias para poder incorporar un algoritmo a la interfaz. Estas fases son:

- Registro
- Inicialización
- Iteración
- Finalización

Cada algoritmo tendrá que estar registrado en la aplicación previamente a su uso.

En la fase de registro, se tendrán que especificar varios aspectos del algoritmo. Éstos son: el nombre identificativo del algoritmo, los diferentes datos de entrada con un nombre determinado para poder reconocerlos posteriormente, los parámetros requeridos, e indicar cuales son las funciones que se corresponden con las tres fases siguientes antes mencionadas. Cabe destacar, que tanto en el caso de indicar los datos de entrada come en el caso de los parámetros se podrá registrar unos valores por defecto, y restricciones en cuanto a valores máximos y/o mínimos.

Registrado ya el algoritmo, el usuario, en la fase de inicialización, podrá indicar los datos que deseará representar, siguiendo unas normas impuestas por la interfaz, tales como el formato de representación (imagen o gráfica), el tipo de dato que se esté tratando, procedencia de los datos (dispositivo OpenCL o host), información acerca de los datos a representar (dimensiones). En esta fase adicionalmente el usuario deberá reservar todos los recursos necesarios para comenzar la iteración del algoritmo.

En la fase de iteración se situarán todas las tareas que deba realizar el algoritmo en cada iteración, devolviendo un valor que aporte información de estado de la ejecución de dicha iteración. Este valor se podrá usar para comprobar errores y detener la ejecución, indicar que se desea continuar iterando o interrumpir porque se ha conseguido el resultado buscado, y para indicar que todo ha ido bien.

La fase de finalización, se encargará de liberar todos los recursos que se han utilizado o reservado para un algoritmo

En todas las fases se controlará el estado de funcionamiento mediante un valor de retorno, y un mensaje de alerta que te indicará el código de error que se haya producido, en caso de fallo.

Una vez completadas estas cuatro etapas por parte del usuario, el algoritmo quedará registrado en la interfaz. Lo cuál, implica la creación de una estructura de datos que almacenará la información correspondiente al algoritmo registrado. En detalle, se guardará el nombre del algoritmo, los parámetros y datos de entrada asociados, los datos que serán representados una vez ejecutada la primera iteración, la localización de las funciones de inicializar, iterar y finalizar correspondientes a las fases anteriormente mencionadas y un indicador de iteración, para dar esa información posteriormente al usuario, a través de la interfaz gráfica.

Al seleccionar un algoritmo de la lista, se detectará el algoritmo correspondiente mediante el uso de un índice que lo identificará para el resto de ejecución de la aplicación.

Para evitar conflictos, los botones, que realicen llamadas a la función iteración del usuario, solamente estarán disponibles cuando el algoritmo se haya iniciado sin errores, por lo que esos elementos de la interfaz gráfica permanecerán bloqueados hasta entonces. Se diseña el mismo comportamiento para el botón que ordena la finalización, y la inicialización en el caso de estar ya iniciado. De esta forma el usuario no podrá pulsar en dichos botones aunque sea consciente de que proporcionará un error al ejecutar primero la función de iterar que la de inicializar.

La interfaz, ya iniciado y habiendo iterado al menos una vez el algoritmo seleccionado, llenará los selectores de visualizaciones, se muestran en la figura 3.2. Corresponderán a los datos fijados o marcados para representación en la fase de iniciación. Al seleccionar cualquiera de los posibles, en uno de los cuatro layouts (distribuciones de representaciones) posibles, se representará, en forma de imagen o gráfica, la opción seleccionada y se mantendrá en esa posición, renovándose automáticamente cuando se itera de nuevo, ahorrando al usuario el trabajo y tiempo de volver a seleccionar esa opción de nuevo.

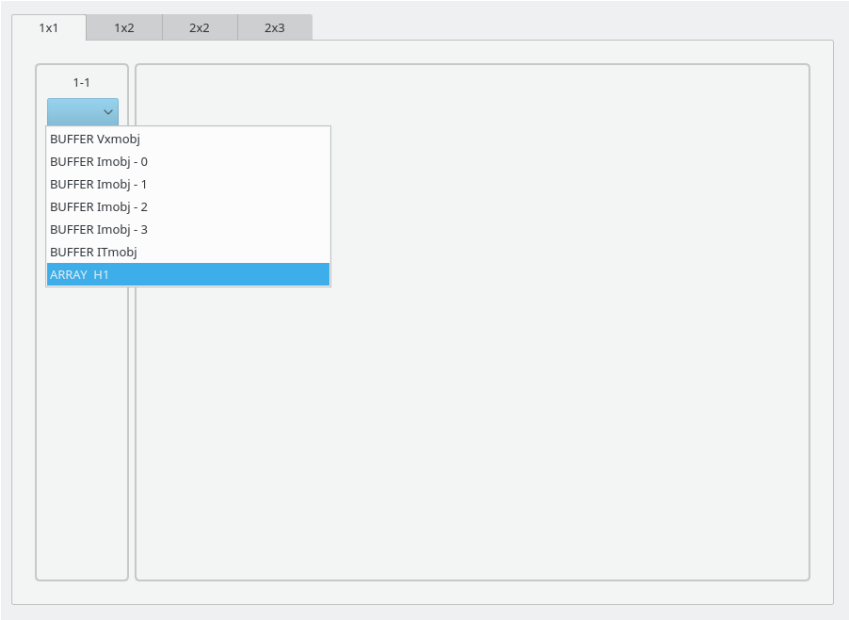

Figura 3.3: Selector de visualizaciones disponibles

Cuando el usuario quiera realizar la opción de iterar, podrá hacerlo de varias maneras. Pulsando el botón correspondiente (Botón Iteración figura 3.1) manualmente cada vez, o si quiere ejecutar varias iteraciones seguidas tendrá que especificar un número finito para lanzar las iteraciones. Adicionalmente tiene otra opción y es lanzando las iteraciones sin fijar un número máximo de ellas, se basará en el código de estado devuelto para continuar o detener el proceso.

A la hora de exportar las representaciones de los datos, se habilitarán dos opciones.

La primera de ellas es manual y requerirá la pulsación del usuario en el botón de guardar, se conseguirá exportar una imagen del layout que esté seleccionado en ese momento y con las representaciones que contenga en él, en una ubicación indicada por el usuario. La segunda opción consistirá en marcar una casilla que indicará al programa que se desea exportar automáticamente todas las visualizaciones disponibles, cada vez que ocurra una iteración.

Cuando se cambia de algoritmo, la interfaz modificará el índice de identificación de algoritmo. Así, se selecciona entre la lista de estructuras de datos correspondiente a cada índice, lo que permite poder cambiar entre algoritmos incluso cuando alguno de ellos esté inicializado o haya iterado alguna vez, no se perderá el progreso de ejecución del mismo al cambiar entre los algoritmos disponibles.

3.3. Implementación

La implementación de la interfaz se ha realizado con las herramientas ofrecidas por Qt. El lenguaje de programación que se ha empleado como base es C/C++. Adicionalmente a C++, se han incorporado dos herramientas más, la biblioteca OpenCV y el lenguaje de programación OpenCL.

La ejecución de la interfaz se realiza gracias a la creación de un objeto de una clase pública denominada *MainWindow*. Será la encargada de iniciar el entorno gráfico y de controlar, a través de sus atributos, métodos y slots, todos los eventos que ocurran en ese entorno.

Para almacenar información, de los diferentes algoritmos y contenidos de los objetos físicos de la interfaz que dependen de la naturaleza del algoritmo, se han empleado estructuras de datos o structs, ordenadas en vectores. El objetivo de usar vectores de estructuras es proporcionar independencia entre algoritmos.

Para permitir una comunicación entre la interfaz y los algoritmos se han creado una funciones específicas que no se encuentran dentro de la clase, y sirven para facilitar al usuario la indicación de los datos de entrada y los parámetros de su algoritmo.

3.3.1. Clase MainWindow

Para explicar el contenido de la clase MainWindow, podemos diferenciar dos partes, los atributos y los métodos de la clase. A su vez los métodos se dividen en funciones públicas y slots privados.

Atributos

Dentro de los atributos se encuentran todas las variables necesarias para controlar el estado de los objetos de la interfaz. Por ejemplo, valores que dictan cuando una representación debe actualizarse en cada iteración, o cuando deben borrarse de la interfaz, el índice de selección de algoritmo y si se encuentra o no inicializado. También se encuentran los punteros que se encargarán de llamar a las funciones principales de un algoritmo (iniciar,

iterar y finalizar) en su botón correspondiente. Se definen vectores para el almacenamiento de objetos de la interfaz. Y finalmente se declaran los buffers y matrices OpenCV de la interfaz que se usarán en la representación de datos.

Métodos

Se comenzará explicando sin entrar en detalles de programación el funcionamiento de cada método. Comenzando por los métodos públicos:

- **int RegisterAlgoritms()**: Una vez iniciada la interfaz, esta será la primera función que se ejecutará. Su cometido es cargar los diferentes algoritmos o módulos en el selector, empleando cada una de las funciones de registro de los diferentes algoritmos implementadas por el usuario.
- **void load_Ivector(struct,int)**: Se encarga de mostrar en la interfaz los parámetros modificables de un algoritmo. Toma una estructura con toda la información de un parámetro y un entero que sirve para controlar el orden de representación.
- **void load_Pvector(struct,int)**: Función similar a la anterior, muestra en el panel central los datos de entrada necesarios.
- **load_values()**: Una vez creados los objetos correspondientes a los parámetros y datos de entrada en la interfaz (las dos funciones anteriores), esta función permite registrar cualquier modificación que se haya producido en esos objetos. Para aclarar un poco más, proporciona la capacidad a la interfaz de mantener unos valores por defecto, disponibles para reiniciar los valores, y unos valores modificados por el usuario al iniciar su algoritmo.
- **void load_algorithm()**: al elegir un algoritmo, se tomará un índice y se dará paso a esta función, que se encargará de ejecutar en orden las tres funciones anteriores. También controlará la interfaz, asignando a los botones de inicializar, iterar y finalizar las funciones correspondientes al algoritmo seleccionado, y bloqueando o desbloqueando ciertos paneles y botones .
- **void fill_layout_list()**: Rellena el selector de representaciones con todas las ofrecidas por el usuario.
- **void empty_layout_list**: Limpia los selectores de representaciones.
- **void show_iteration(struct,int)**: Función principal de representación de los datos, recoge como entrada una estructura con la información de los datos a representar (dependiendo del índice del selector de representaciones), y un entero para decidir el espacio donde se representará la imagen o gráfica. Se han usado aquí las matrices y funciones que proporciona OpenCV para almacenar los datos, y realizar operaciones sobre ellos (normalización, transformación del tipo de dato, operaciones sobre datos de tipo complejo ...).

- **void comboBoxSelected()**: Con el objetivo de representar automáticamente, lo que ya está seleccionado, en cada iteración esta función guarda los índices de todas las representaciones de todos los layouts.
- **void checkAndShow()**: Función que ejecuta el objetivo de la función anterior, verifica índices y decide si debe representar o no.
- **void exportIteration()**: Si el usuario solicita exportar y guardar automáticamente las representaciones, esta función se encargará de ello usando un formato .png.
- **void clear_labels()**: Limpiará las representaciones, tanto las imágenes como las gráficas.
- **void clearAlgorith()**: Se encarga de limpiar el panel central.
- **void disableWidget()**: En ciertas situaciones conviene bloquear los elementos de la interfaz (ejecución en bucle de las iteraciones), para que no se produzcan errores, y ese es el objetivo de esta función.

Hasta aquí el conjunto de métodos públicos. Para finalizar la explicación de la clase MainWindow, queda por indicar los slots o funciones asignadas a un cierto evento de la interfaz. Se pondrán tres ejemplos que aclaren los tres tipos de eventos que se han usado para esta interfaz.

- **void on_listWidget_itemSelectionChanged()**: Se ejecuta solamente cuando recibe una señal desde el selector de algoritmos, y esa señal corresponde a una nueva selección. Se encargará de modificar el índice de algoritmo.
- **on_pushButton1_released()**: La señal que recibe es una pulsación sobre el botón de inicializar algoritmo. Recogerá los datos del panel central (parámetros y datos) y llamará a la función de usuario correspondiente a inicializar algoritmo.
- **on_comboBox_activated(int)**: Se recibe una señal que informa de la activación de una representación del selector, es decir que a través de una entrada de tipo entero nos indica que datos se quieren representar. Se encargará de llamar a la función *show_iteration* anteriormente explicada.

3.3.2. Estructuras de Información

Para el almacenamiento de datos, se han definido varias estructuras. Dos estructuras sencillas y básicas donde se recoge información de los parámetros y datos de entrada indicados por el usuario en la fase de registro. Otra estructura un poco menos sencilla que contiene los parámetros y datos de entrada modificados desde la interfaz. Es decir los valores por defecto o las modificaciones. Y otra estructura dedicada a las opciones de representación de datos.

Todas ellas formarán una estructura principal que se denomina *Algoritmo*. En la cuál irán incluidas las localizaciones o punteros de las funciones principales del usuario (iniciar, iterar, finalizar). Adicionalmente, se ofrece un puntero genérico para que el usuario pueda crear sus propias estructuras y así se le da la posibilidad de compartir memoria entre sus funciones, y un indicador de iteración actual. También incluye un identificador de algoritmo para el usuario en forma de nombre.

Se define una última estructura auxiliar para el usuario y totalmente opcional, donde se declaran todas los objetos básicos para la implementación de un algoritmo usando OpenCL.

3.3.3. Funciones de Interacción

Son una pareja de funciones sobrecargadas (mismo nombre,distinto número de parámetros), que permiten al usuario indicar que parámetros y datos de entrada requiere su algoritmo, de una forma sencilla. El objetivo de crear sobrecarga de funciones, es dar una mayor comodidad al usuario para poder introducir valores por defecto. Se presentan las dos funciones en su declaración con mayor número de parámetros.

- **void load_param(nombre, tipo,valor_defecto,valor_mínimo,valor_máximo,Algoritmo*)**: De entre todos los parámetros disponibles en esta función sobrecargada, los obligatorios son el nombre, tipo de dato y el puntero a la estructura *Algoritmo* que corresponda. Los demás, el valor por defecto, el valor mínimo y el valor máximo son opcionales. El funcionamiento de esta función es muy básico, y simplemente agregará parámetros a su *Algoritmo*.

- **void load_path(nombre,valor_defecto,Algoritmo*)**: El único parámetro opcional en este tipo de función es el valor por defecto. El funcionamiento es equivalente a la anterior, agregará los nombres de los datos de entrada necesarios al *Algoritmo*.

Capítulo 4

Construcción de Módulos para la Aplicación

En este capítulo se explicará paso a paso, con un ejemplo práctico, como se adapta un algoritmo iterativo a la interfaz.

4.1. Proceso de Construcción del Módulo

El módulo usado para ejemplificar se llama GroupwiseRegistration. Es una versión simplificada de un algoritmo iterativo que procesa una serie de imágenes (un vídeo) de un corazón en movimiento, deformándolas para congelar el movimiento. Se ha escogido este ejemplo, ya que procesa los datos en la GPU y sirve de prototipo de uso de OpenCL en la interfaz.

Antes de comenzar la adaptación, hay que diferenciar tres etapas de un algoritmo iterativo. Son fácilmente distinguibles, ya que suelen seguir este patrón: una fase inicial, donde se fijan las variables y parámetros, se leen los datos de entrada, se reservan los recursos que se usarán, y se ejecutan todas las operaciones necesarias para preparar el proceso iterativo. Una fase de iteración donde se encuentran todas las tareas que se van a realizar indefinidamente hasta que un cierto criterio determine que ya no es necesario ejecutarlas más veces. Y una fase final donde se comprobarían los resultados finales y liberan los recursos empleados. Como estamos usando esta interfaz para comprobar los resultados, en forma de visualizaciones, la etapa final simplemente se encargará de liberar los recursos.

Cuando se hayan distinguidos estas tres fases, se puede empezar a adaptar. El usuario definirá cuatro funciones siguiendo una norma, en cuanto a los nombres que se le dan a estas funciones y su declaración (valores de entrada, y de retorno). Se recomienda nombrar las funciones con el nombre del algoritmo en cuestión y una palabra identificativa a la función, para evitar posibles problemas causados por la declaración de varias funciones con el mismo nombre. Las cuatro funciones a definir siguiendo esta norma son:

- **int NombreAlgoritmo_Registrar(VectorAlgortimos &vec_algo)**: devuelve

un entero, y recibe como referencia, al argumento, el vector de algoritmos donde se encuentran todos los registrados anteriormente (vector donde insertará el *Algoritmo* que va a crear).

- **int NombreAlgoritmo_Inicializar(Algoritmo *algo)**: devuelve un entero y recibe la referencia del *Algoritmo* asignado a este algoritmo.
- **int NombreAlgoritmo_Iterar(Algoritmo *algo)**: devuelve un entero y recibe la referencia del *Algoritmo* asignado a este algoritmo.
- **int NombreAlgoritmo_Finalizar(Algoritmo *algo)**: devuelve un entero y recibe la referencia del *Algoritmo* asignado a este algoritmo.

La primera función se encargará de registrar el algoritmo en la interfaz.

En este punto, antes de especificar que debe contener cada función anterior, es necesario editar el fichero *modules.cpp*, donde se incluirá un fichero de cabecera con las declaraciones de estas funciones y se llamará a la recién creada función de registrar, de la siguiente forma: `int status = NombreAlgoritmo_Registrar(this->vector_algoritmos)`.

Una vez finalizado ese paso, podemos comenzar a completar las cuatro funciones anteriores.

En la función de registrar será necesario crear una estructura *Algoritmo* y almacenar en ella la siguiente información:

- Nombre: identificador del algoritmo.
- Datos de entrada: usando las funciones *load_path*.
- Parámetros de entrada: usando las funciones *load_param*. Los tipos de variables disponibles para estos parámetros están reflejados en la documentación A.2.1.
- Nombres dados a las funciones de inicializar, iterar y finalizar.

El paso final de esta función será añadir la estructura *Algoritmo* recién creada, al vector de algoritmos que se pasa por referencia. En la figura 4.1, se puede ver la función registrar correspondiente al algoritmo GroupwiseRegistration.

Para compartir datos entre las funciones de inicialización, iteración y finalización se recomienda crear una estructura donde se recojan todas las variables simples (variables que tomen sólo un valor), los punteros a los objetos de clases, los punteros a los diferentes buffers necesarios y en caso de estar usando OpenCL se aconseja utilizar la estructura *cl_app* que proporciona la interfaz.

Pasamos a ver lo necesario de la función de inicio. Si no se desea crear una estructura global para el almacenamiento de datos, se puede usa el puntero genérico que incorpora la estructura *Algoritmo*. En esta función se recogen los paths(rutas) de los datos de entrada y los valores de los parámetros registrados anteriormente, para ello *Algoritmo* posee otra estructura denominada *datos_entrada* donde se encuentran los paths de datos y los diferentes tipos de parámetros clasificados (A.2.2, para entender como se organizan). El resto

```
//Registrar algoritmo en la interfaz
int GroupwiseRegistration_Register(VectorAlgoritmos &vec_algo){
    //Creación estructura
    Algoritmo my_algo;

    //Nombre del algoritmo
    my_algo.nombre = "Groupwise Registration";

    //Cargar paths de datos de entrada y kernel
    load_path("Data:",
    "/home/mario/Dropbox/Universidad/TFG/QT/Version10/Modules/GroupwiseRegistration
    /data/dataset_02_SA_288x288x30.bin", &my_algo);
    load_path("Kernel:",
    "/home/mario/Dropbox/Universidad/TFG/QT/Version10/Modules/GroupwiseRegistration
    /cl/GroupwiseRegistrationKernels.cl", &my_algo);

    //Cargar los parametros necesarios
    load_param("Nº Filas:", T_INT, "288", &my_algo);
    load_param("Nº Columnas:", T_INT, "288", &my_algo);
    load_param("Nº Frames:", T_INT, "30", &my_algo);
    load_param("Dp_def:", T_INT, "4", &my_algo);//Distancia entre puntos de control
    load_param("maxiter:", T_INT, "50", &my_algo);//Iteraciones Maximas

    //Funciones de inicializar, iterar y finalizar
    my_algo.fp_init    = GroupwiseRegistration_Init;
    my_algo.fp_iterate = GroupwiseRegistration_Iterate;
    my_algo.fp_finish  = GroupwiseRegistration_Finish;

    //Añadir al vector de algortimos este algoritmo
    vec_algo.push_back(my_algo);
    return 0;
}
```

Figura 4.1: Función de Registro del algoritmo GroupwiseRegistration

de tareas de inicialización dependerá de la naturaleza del algoritmo. Para poder usar la capacidad de visualización de la interfaz, es necesario definir en esta función una estructura del tipo *show_data_type* que se utilizará para rellenar información sobre los datos a representar. Y se incluirá, cada una de estas estructuras (una por cada visualización) en el vector, que se encuentra en *Algoritmo*, denominado *datos_a_representar*. Consultar A.2.2 para contemplar las diferentes opciones de visualización.

Para ejemplificar lo mencionado en el párrafo anterior, se muestra en la figura 4.2, como se especifica la representación en forma de imagen de los datos de un buffer localizado en la memoria del dispositivo OpenCL.

En la función de iterar se implementan las tareas necesarias para cada algoritmo, la única peculiaridad a tener en cuenta es el valor de retorno de esta función. Si el usuario desea, que al pulsar el botón *Lanzar Iteraciones*, el algoritmo continúe iterando, esta función deberá devolver un uno. En cambio, si quiere detener la ejecución del lanzamiento de iteraciones, deberá retornar un valor mayor que uno. Mismo criterio para el botón *Iterar*, aunque sea manual (pulsación del botón, equivale a una iteración).

Para terminar, se debe implementar la función de finalizar, para liberar los recursos empleados (buffers, objetos, estructuras ...), y el puntero genérico (si se ha utilizado). No es necesario liberar nada más relacionado con la estructura *Algortimo* (las representaciones,

```
//Creación de la estructura a rellenar
show_data_type show_struct;

//Nombre de la visualización
show_struct.name = "BUFFER ITmobj";
//Tipo de dato, en este caso tipo CL, que implica representación desde
//memoria del dispositivo OpenCL
show_struct.data_type = TSD_CL;
//Forma de representación, en este caso una imagen
show_struct.show_type = TD_IMAGE;
show_struct.size.clear();
//Indicar que tamaño va a tener la imagen
show_struct.size.push_back(Nx1);
show_struct.size.push_back(Nx2);
//Al ser tipo CL, es necesario rellenar una estrucutra adicional (data_cl)
//Tamaño del buffer donde están alojados los datos
show_struct.data_cl.cl_buffer_size = Nx1*Nx2;
//Dirección del buffer
show_struct.data_cl.buffer        = my_gwreg->cl.buffers[1];//buffer ITmobj
//Cola asignada al dispositivo OpenCL
show_struct.data_cl.command_queue = my_gwreg->cl.queue;
//Tipo de los datos del buffer, en este caso son floats
show_struct.data_cl.cl_data_type  = cl_FLOAT;
//Añadir al vector de estructuras de representación
algo->datos_a_representar.push_back(show_struct);
```

Figura 4.2: Ejemplo estructura de visualización de los datos de un buffer OpenCL

por ejemplo), de eso se encargará la aplicación.

Siguiendo estos paso se conseguirá adaptar el algoritmo iterativo a la interfaz y se podrán visualizar los datos deseados, evitando todo el proceso que sería necesario realizar en un entorno de desarrollo tradicional para poder hacer representaciones de los datos en GPU y poder así, efectuar las tareas de depuración de una forma más rápida y sencilla.

Adicionalmente a este capítulo, en el contenido de este CD, se pueden encontrar ejemplos que le servirán de guía, o le ayudarán a comprender como es el proceso de adaptación en esta interfaz sino se han despajado las dudas con esta sección.

Capítulo 5

Conclusiones y Líneas Futuras

Este capítulo contiene las conclusiones extraídas durante el desarrollo de la aplicación. Seguidamente, se recogen las posibles vías de ampliación o modificación que la aplicación podría seguir.

5.1. Conclusiones

La finalidad o propósito de este trabajo es la programación de una aplicación dedicada a la visualización de datos multidimensionales durante la ejecución de un algoritmo iterativo cualquiera, implementado en GPU. Se ha desarrollado una interfaz gráfica que permite comprobar, de una forma visual, el correcto funcionamiento de un algoritmo.

Partiendo del objetivo básico, se han ido añadiendo funcionalidades adicionales, tales como la exportación de las visualizaciones, la introducción de gráficas como medio de representación complementario, o la modificación dinámica de parámetros y datos de entrada del algoritmo.

Por otro lado, se han encontrado dificultades en la implementación que limitan en cierta forma al sistema, por ejemplo la extensa variedad de modelos de color,y su capacidad para representar imágenes usando más o menos canales de información, lo que provocó la adopción de un modelo más sencillo, el modelo en blanco y negro (o escala de grises) para la representación de imágenes, o la forma de indicar que datos se quieren visualizar no es del todo cómoda de usar a primera vista.

A pesar de las restricciones que puedan ocasionar estos obstáculos, se ha demostrado que la visualización de datos mediante el uso de esta interfaz, se realiza de un modo más ágil, ordenado y menos tedioso.

Para finalizar, me gustaría hacer una conclusión a nivel más personal. A lo largo de todo el desarrollo del trabajo, he ido aprendiendo a usar varias tecnologías y lenguajes de programación. Desde el diseño de interfaces gráficas con Qt, hasta la implementación de ejemplos de algoritmos iterativos en GPU mediante OpenCL, ha supuesto un desafío, dado que todo ello era terreno desconocido. De las herramientas empleadas, destacaré OpenCV, no tenía conocimiento de ella y me parece que es una utilidad realmente buena y me ha sorprendido con creces.

5.2. Líneas Futuras

Como se ha mencionado antes, la aplicación tiene algunas limitaciones, que pueden ser superadas mediante la adición de nuevas funcionalidades en próximas versiones, tales como:

- Aumentar los modelos de color soportados y dar más flexibilidad en el número de canales a la hora de representar datos, ya que actualmente solo admite un canal (escala de grises, de negro a blanco, con valores desde el 0 hasta el 255).
- Implementar funciones que faciliten la especificación de los datos que se desean representar. Y modificar o buscar una alternativa a la estructura que ofrece los datos y parámetros de entrada al usuario.
- Permitir en las representaciones de gráficas, e incluso en la imágenes, realizar un zoom mediante una herramienta estilo lupa para poder ver secciones de la representación a un tamaño mayor. Junto con esta lupa, otra utilidad que permitiera recoger un valor concreto de la representación mediante una pulsación en un punto cualquiera de ella. Esto sería muy beneficioso para el caso de las gráficas.

Apéndice A

A.1. Manual de Usuario

En esta sección se explican los objetos que incorpora la interfaz gráfica.

Partiendo de una ejecución inicial de la interfaz, dispondremos de una ventana con un selector habilitado, como se muestra en la figura A1.

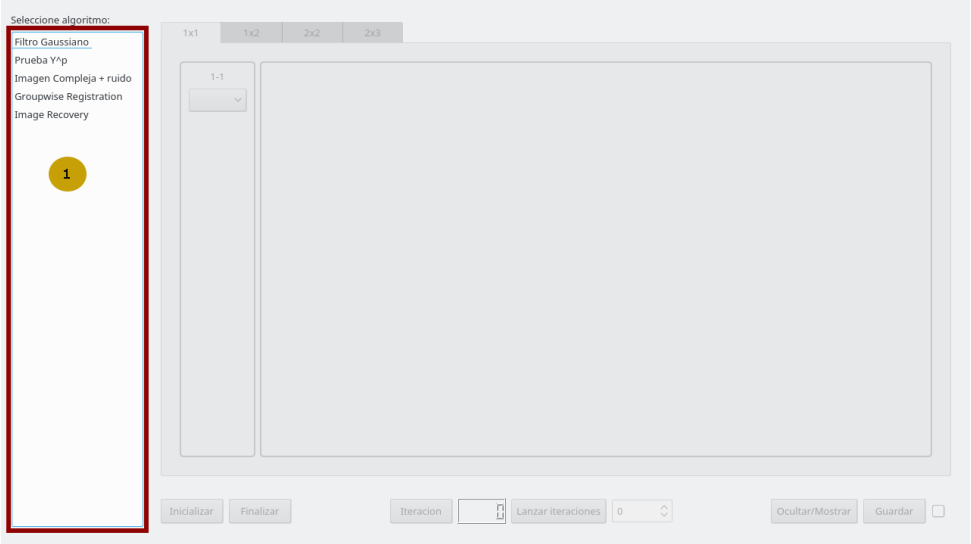

Figura A.1: Ventana inicial

El primer paso será escoger el algoritmo deseado mediante un clic en el nombre que lo identifica. Al seleccionar una opción de las disponibles aparece un panel justo a la derecha del selector, que contiene los datos de entrada y parámetros necesarios.

En orden de numeración de la figura A.2:

- 1: Panel que contiene los elementos.
- 2: Datos de entrada, en la interfaz aparece para cada dato, un nombre identificativo, un *text edit* donde se puede editar la ruta del dato manualmente o mediante el botón

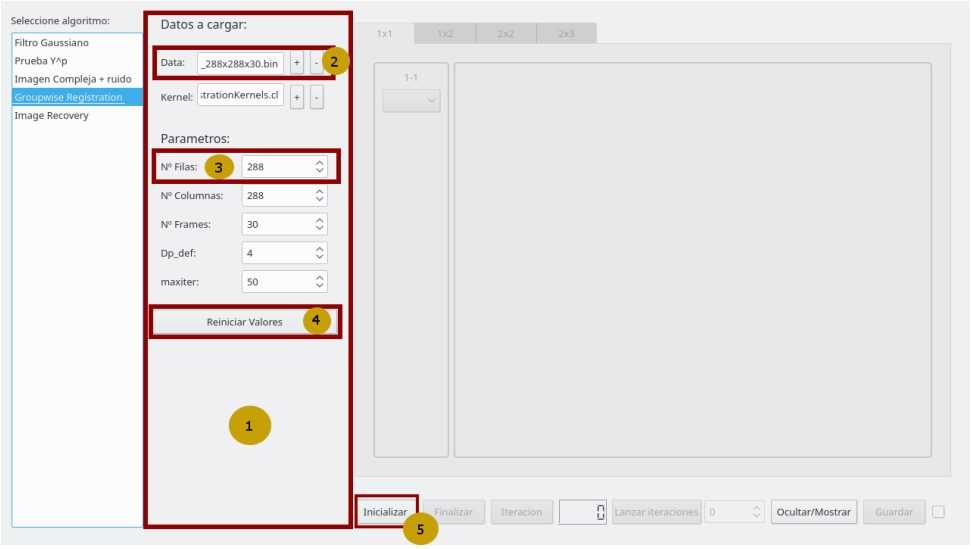

Figura A.2: Algoritmo seleccionado

con el símbolo +, que abrirá un explorador de ficheros. El botón con el símbolo -, deja en blanco el *text edit*.

- 3: Parámetros, para cada parámetro, se especifica el nombre, un valor por defecto (si lo tiene) y el valor se puede editar haciendo clic y escribiendo el número o mediante los controles del objeto.
- 4: Botón Reiniciar Valores, restaura los valores por defecto.
- 5: Botón Inicializar, ejecutar la función de inicio correspondiente al algoritmo, si no se produce ningún error desbloquea la siguiente parte de la interfaz, y recoge la información introducida en el panel central.

Al inicializar se habilitan el resto de botones del panel inferior, en la figura A.3 se pueden ver los diferentes botones.

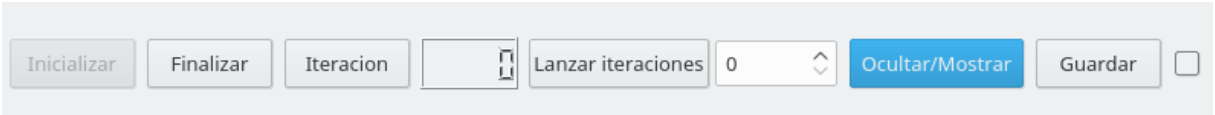

Figura A.3: Panel de botones

En orden, de izquierda a derecha, en la figura A.3 aparecen:

- Botón Inicializar bloqueado, debido a que el algoritmo está iniciado.
- Botón Finalizar, corresponde a la función de finalizar del algoritmo.

- Botón Iteración, corresponde a la función de iterar del algoritmo. Ejecuta una iteración.
- Display, contador de iteraciones. Refleja la iteración actual en la que se encuentra el algoritmo.
- Botón Lanzar iteraciones, corresponde a la función iterar del algoritmo. Itera un número concreto de veces o indefinidamente hasta nueva orden.
- Fijador del número máximo de iteraciones que ejecuta el botón Lanzar iteraciones.
- Botón Ocultar/Mostrar, esconde o muestra el panel central.
- Botón Guardar, abre un explorador para que el usuario guarde una captura de lo que se encuentre actualmente representado.
- Checkbox o casilla, cuando esté marcada se guardarán automáticamente en el directorio Outputs/NombreAlgoritmo las imágenes de todas las representaciones disponibles para ese algoritmo cada vez que se itere.

Al iterar al menos una vez se desbloquearán las pestañas de los layouts (1x1, 1x2, 2x2 y 2x3), y los selectores de representaciones de cada layout. En la figura A.4 se pueden ver las diferentes opciones escogidas para el algoritmo GroupwiseRegistration.

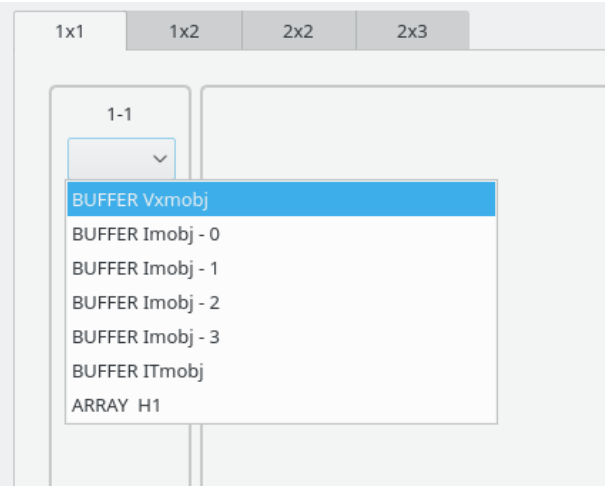

Figura A.4: Selector representaciones GroupwiseRegistration

Al elegir cualquiera de las opciones, se representará en su respectiva posición dependiendo de la pestaña escogida. El número que aparece encima de cada selector indica la posición respecto al layout (fila-columna). Para finalizar, en la figura A.5 se puede ver un ejemplo con varias representaciones y el panel central escondido.

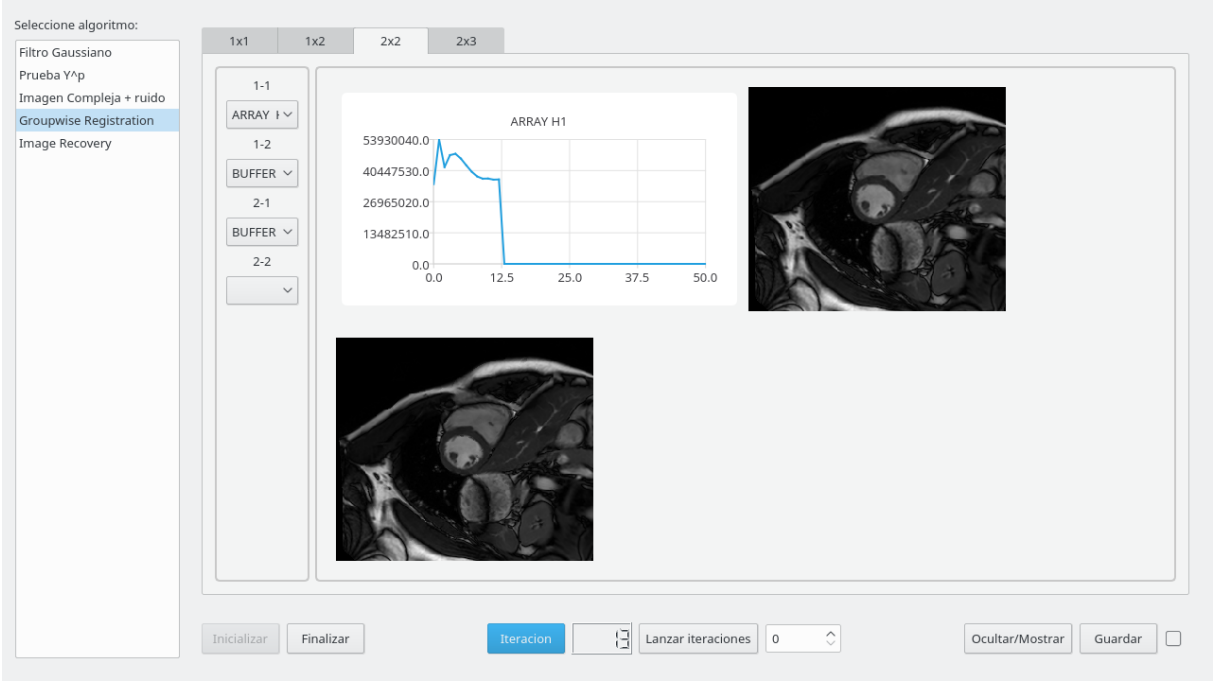

Figura A.5: Varias representaciones GroupwiseRegistration, con panel central oculto

A.2. Documentación

En esta sección se detalla toda la información relevante al usuario, en términos de métodos y estructuras que utilizará y cumplimentará.

A.2.1. Métodos

load_path()

Definiciones de los métodos sobrecargados load_path, sirven para especificar un dato de entrada de un algoritmo, se llaman desde la función de registro del algoritmo:

- `void load_path(std::string nombre,Algoritmo*);`
- `void load_path(std::string nombre,std::string valor_defecto,Algoritmo*);`

Definiciones de los métodos sobrecargados load_param, sirven para especificar un parámetro de un algoritmo, se llaman desde la función de registro del algoritmo:

- `void load_param(std::string nombre,Tipo,Algoritmo*);`
- `void load_param(std::string nombre,Tipo,std::string valor_defecto,Algoritmo*);`
- `void load_param(std::string nombre,Tipo,std::string valor_defecto,std::string valor_minimo,Algoritmo*);`

- `void load_param(std::string nombre,Tipo,std::string valor_defecto,std::string valor_minimo,std::string valor_maximo,Algoritmo*);`

Siendo *Tipo* un enum, que distingue los diferentes tipos de parámetro:

- `T_STRING`, tipo texto.
- `T_INT`, tipo entero.
- `T_FLOAT`, tipo float.
- `T_DOUBLE`, tipo double.
- `T_COMPLEX_RI`, tipo complejo, parte real (double), y parte imaginaria (double). En el campo valor_defecto, se especifican las diferentes partes usando una coma. Ejemplo: "4.3,3.4"equivaldría al complejo 4.3 + i*3.4.
- `T_COMPLEX_P`, tipo complejo, módulo (double) y argumento (double). En el campo valor_defecto, se especifican las diferentes partes usando una coma. Ejemplo: "2,3.14"equivaldría al complejo 2*exp(i*3.14).

A.2.2. Estructuras

datos_entrada

Estructura *data_struct* que organiza los paths de datos de entrada, y los valores parámetros, por tipo y orden de registro. Esto significa que dependiendo del orden en el que se han registrado los diferente parámetros y paths de datos, se clasificarán, en ese mismo orden, dependiendo de su tipo:

- `std::vector <std::string> data_string;` vector de valores del parámetro *Tipo* `T_STRING`.
- `std::vector <float> data_float;` vector de valores del parámetro *Tipo* `T_FLOAT`.
- `std::vector <double> data_double;` vector de valores del parámetro *Tipo* `T_DOUBLE`.
- `std::vector <int> data_int;` vector de valores del parámetro *Tipo* T_INT.
- `std::vector <std::complex<<double>> data_complex;` vector de valores del parámetro *Tipo* `T_COMPLEX_RI` y `T_COMPLEX_P` se guardan ambos como complejos de parte real e imaginaria.
- `std::vector <std::string> path_string;` vector de rutas de los datos.

datos_a_representar

Es un vector de estructuras *show_data_type*, donde se especifica que se quiere representar. La estructura *show_data_type* consta de:

- `std::string name;` Nombre que tendrá la representación
- `std::vector<int>size;` Sirve para definir el alto y ancho (filas y columnas) de la imagen o del tamaño del vector para los plots. En caso de ser imagen es necesario los dos tamaños, en caso de ser plot solo un tamaño.
- `TipoShowData data_type;` Define el tipo de dato representable, consultar más adelante *TipoShowData*.
- `TipoDisplay show_type;` Define el tipo de representación (imagen o plot) `TD_IMAGE` o `TD_PLOT`
- `std::string axis_names[2];` En plots, para dar nombre a los ejes, primero el X y luego el Y
- `void* data_ptr;` Puntero a los datos para los cuatro primeros tipos *TipoShowData*.
- `void* dat_ptr_aux = nullptr;` Puntero a los datos que se usarán para el eje X de los plots, si es null, se tomará como eje X el vector 0:1:longitud_datos - 1
- `show_data_complex data_complex;` Estructura que es necesaria para representaciones de datos complejos, quinta opción de *TipoShowData*
- `show_data_cl data_cl;` Estructura que es necesaria para representaciones de datos usando OpenCL, sexta opción de *TipoShowData*

Siendo *TipoShowData* un enum, que distingue los diferentes tipos de datos representables:

- `TSD_DOUBLE`, se usa en la estructura anterior para indicar que los datos apuntados por data_ptr son de tipo double.
- `TSD_FLOAT`, se usa en la estructura anterior para indicar que los datos apuntados por data_ptr son de tipo float.
- `TSD_INT`, se usa en la estructura anterior para indicar que los datos apuntados por data_ptr son de tipo int.
- `TSD_UCHAR`, se usa en la estructura anterior para indicar que los datos apuntados por data_ptr son de tipo unsigned char.
- `TSD_COMPLEX`, se usa en la estructura anterior para indicar que los datos serán complejos, por lo que será necesario rellenar otra estructura adicional, data_complex.
- `TSD_CL`, se usa en la estructura anterior para indicar que los datos se leerán usando OpenCL, por lo que será necesario rellenar otra estructura adicional, data_cl.

La estructura data_complex es de tipo *show_ data_ complex* y almacena la siguiente información:

- `void* complex_pointer = nullptr;` Puntero a los datos complejos
- `TipoOperacionCompleja complex_operation = TOC_REALvIMAG;` Tipo de operación que se quiera realizar sobre los datos complejos
- `uint complex_data_array_size;` Tamaño complejo del buffer que contiene los datos (ejemplo, 8 números complejos en el buffer significa que hay 8 reales y 8 imaginarios, es decir 16. El tamaño que ser debe indicar aquí es 16)
- `TipoComplejo complex_data_type;` `TC_DOUBLE` o `TC_FLOAT`, los datos complejos pueden ser double o float

Siendo *TipoOperacionCompleja* un enum, que distingue las diferentes operaciones posibles sobre los complejos

- `TOC_REAL`, solo se cogen los datos de la parte real, para su representación.
- `TOC_IMAG`, solo se cogen los datos de la parte imaginaria, para su representación
- `TOC_ANGLE`, se obtiene el ángulo de cada complejo
- `TOC_ABS`, se obtiene el valor absoluto de cada complejo
- `TOC_REALvIMAG`, representa el valor real frente al imaginario en un plot.

La estructura data_cl es de tipo *show_ data_ cl* y almacena la siguiente información:

- `cl_command_queue command_queue;` Puntero a la cola
- `cl_mem buffer;` OpenCL buffer donde se encuentran los datos
- `uint cl_buffer_size;` tamaño del buffer sin multiplicar por el tamaño de tipo de dato
- `uint offset = 0;` indicar si se desea offset, sin multiplicar por el tamaño de tipo de dato(sin el sizeof(tipodato))
- `TipoCL cl_data_type;` Tipo de dato que se encuentra en el buffer
- `TipoOperacionCompleja cl_complex_operation = TOC_REALvIMAG;` Solamente aplicable para la opción quinta y sexta de *TipoCL*

Siendo *TipoCL* un enum, que distingue los diferentes tipos de dato que se almacenen en el buffer OpenCL a representar:

- `cl_DOUBLE`, tipo de dato double para OpenCL
- `cl_FLOAT`, tipo de dato float para OpenCL
- `cl_INT`, tipo de dato entero para OpenCL
- `cl_UCHAR`, tipo de dato unsigned char para OpenCL
- `cl_COMPLEX_DOUBLE`, tipo de dato complejo double para OpenCL
- `cl_COMPLEX_FLOAT`, tipo de dato complejo float para OpenCL

Algoritmo

La estructura *Algoritmo* almacena todo los datos correspondientes al algoritmo que la creó y registró en la interfaz. Contiene la siguiente información para el usuario:

- `std::string nombre;` Nombre asignado a este algoritmo
- `show_data_vector datos_a_representar;` Vector donde se guardan los datos que se quieren representar
- `data_struct datos_entrada;` Estructura donde se encuentran todos los valores de los parámetros y paths.
- `int (*fp_init) (Algoritmo*) = nullptr;` Puntero a la función de inicialización
- `int (*fp_iterate)(Algoritmo*) = nullptr;` Puntero a la función de iteración
- `int (*fp_finish) (Algoritmo*) = nullptr;` Puntero a la función de finalización
- `void* void_pointer = nullptr;` Puntero genérico
- `uint icounter = 0;` Contador de iteraciones

cl_app

Estructura auxiliar para trabajar con OpenCL, el usuario puede utilizarla o no.

- `cl_platform_id platform;`
- `cl_context context;`
- `cl_command_queue queue;`
- `cl_device_id device;`
- `cl_program program;`
- `cl_kernel *kernels;`

- `cl_int num_kernels;`
- `cl_mem *buffers;`
- `cl_int num_buffers;`

A.3. Contenido del CD

Adicionalmente a la memoria, se entrega todo el código correspondiente a la aplicación realizada. Incluyendo varios ejemplos, independientes entre sí, que demuestran el funcionamiento de la interfaz y sirven de ayuda para quién quiera utilizar esta herramienta.

La estructura de directorios que se entrega en este CD es la siguiente:

- En el mismo directorio que la memoria, se encuentra el código de la aplicación en un directorio denominado *Version10*.
- Dentro del directorio anterior, el trabajo se encuentra dividido en dos subdirectorios. En el primero de ellos, en *src* se aloja todo el código fuente correspondiente a la interfaz, su diseño gráfico e implementación. En el mismo nivel que el subdirectorio anterior está el directorio *Modules*, que contendrá los módulos independientes de cada algoritmo, en su propio directorio cada uno, es por eso que cada ejemplo entregado con este trabajo tiene dentro de este directorio, el suyo propio.

Bibliografía

[1] Ronal L. Graham, Donal E. Knuth, Oren Patashnik. *Concrete Mathematics*. Addison Wesley, 1989.

[2] Ivar Jacobson, Grady Booch, James Rumbaug. *El Proceso Unificado de Desarrollo de Software*. Addison Wesley, 1999.

[3] Wilbert O. Galitz. *The Essential Guide to User Interface Design, An Introduction to GUI Design Principles and Techniques*. John Wiley & Sons, Inc, 2002.

[4] The Qt Company. *About Qt*. URL: `http://wiki.qt.io/About_Qt`. Consultado: Enero 2018.

[5] Advanced Micro Devices, Inc. *Cómputo de GPU*. URL: `http://www.amd.com/es-xl/products/graphics/server/gpu-compute`. Consultado: Enero 2018.

[6] Peng Dua, Rick Webera, Piotr Luszczeka, Stanimire Tomova, Gregory Petersona, Jack Dongarra. *From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming*. University of Tennessee Knoxville & University of Manchester. Abril 2011.

[7] The Khronos™ Group Inc *The OpenCL Specification v 1.2*. URL: `https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf` Consultado: Enero 2018.

[8] NVIDIA Corporation. *Procesamiento Paralelo CUDA*. URL: `www.nvidia.es/object/cuda-parallel-computing-es.html`. Consultado: Enero 2018.

[9] The Khronos™ Group Inc. *OpenCL Overview*. URL: `https://www.khronos.org/opencl/`. Consultado: Enero 2018.

[10] Aaftab Munshi, Benedict R. Gaster, Timothy G. Mattson, James Fung, Dan Ginsburg. *OpenCL Programming Guide*. Addison Wesley, 2012.

[11] Gary Bradski, Adrian Kehler. *Learning OpenCV*. O'Reilly Media, Inc. 2008.

[12] OpenCV Team. *About*. URL: `https://opencv.org/about.html`. Consultado: Enero 2018.

[13] T. S. Huang. *Computer Vision: Evolution and Promise*. University of Illinois at Urbana-Champaign. 2003.

[14] OpenCV Team. *Smoothing Image Tutorial* URL: `https://docs.opencv.org/3.4.0/dc/dd3/tutorial_gausian_median_blur_bilateral_filter.html` Consultado: Enero 2018.

[15] OpenCV Team. *Sobel Derivatives Tutorial* URL: `https://docs.opencv.org/3.4.0/d2/d2c/tutorial_sobel_derivatives.html` Consultado: Enero 2018.