Instituto Tecnológico de Buenos Aires

Tesis de Grado

Validación de Sistema de Visualización y Alineación Volumétrica para selección de alo-injertos óseos en huesos largos.

Autor:
Facundo Mercado
Legajo: 56067

Directores:
Lucas E. Ritacco, M.D., PhD.
Axel V. Mancino, Ing.

3 de junio de 2020

Índice

1. Introducción

En los siguientes incisos se presenta el marco teórico que sustenta el proyecto, así como también se menciona el estado del arte de las tecnologías abordadas.

1.1. Cirugía Asistida por Computadora

La **Cirugía Asistida por Computadora** (CAS) es una disciplina basada en una variedad de tecnologías y en distintas fuentes de información [1]. Ésta consiste en un procedimiento, en el cual imágenes digitales de una persona, obtenidas a través de un dispositivo de generación de imágenes médicas (por ejemplo la Tomografía Computada o "TC") y una computadora, son utilizadas para la recreación virtual tridimensional de regiones anatómicas. Los cirujanos luego utilizan este modelo durante una intervención quirúrgica como una guía para "navegar" la zona de interés en el paciente a tratar y completar la cirugía de manera precisa y segura.

Con esta disciplina se han logrado mejorar los resultados de una intervención quirúrgica, en particular en los siguientes aspectos [2] [3]:

- Realizar incisiones y cortes pequeños.
- Realizar procedimientos quirúrgicos mínimamente invasivos.
- Localizar la lesión eficientemente, resultando en un tiempo de operación más corto.
- Conocer las estructuras críticas y evitarlas.
- Planificar y simular el mejor camino para ejecutar la cirugía.

CAS incluye dos áreas [2] denominadas **"Planeamiento Virtual Preoperatorio"** (Virtual Preoperative Planning o VPP en inglés) y **"Navegación Virtual Intraoperatoria"** (Intraoperative Virtual Navigation o IVN en inglés). Posibles aplicaciones de estas áreas incluyen, la medición de márgenes oncológicos en 3 dimensiones, localización de tumores y aplicación de resecciones, asegurándose de preservar la estructura anatómica en el proceso (figura 1, página 4). Estas dos áreas serán detalladas y caracterizadas en los siguientes incisos.

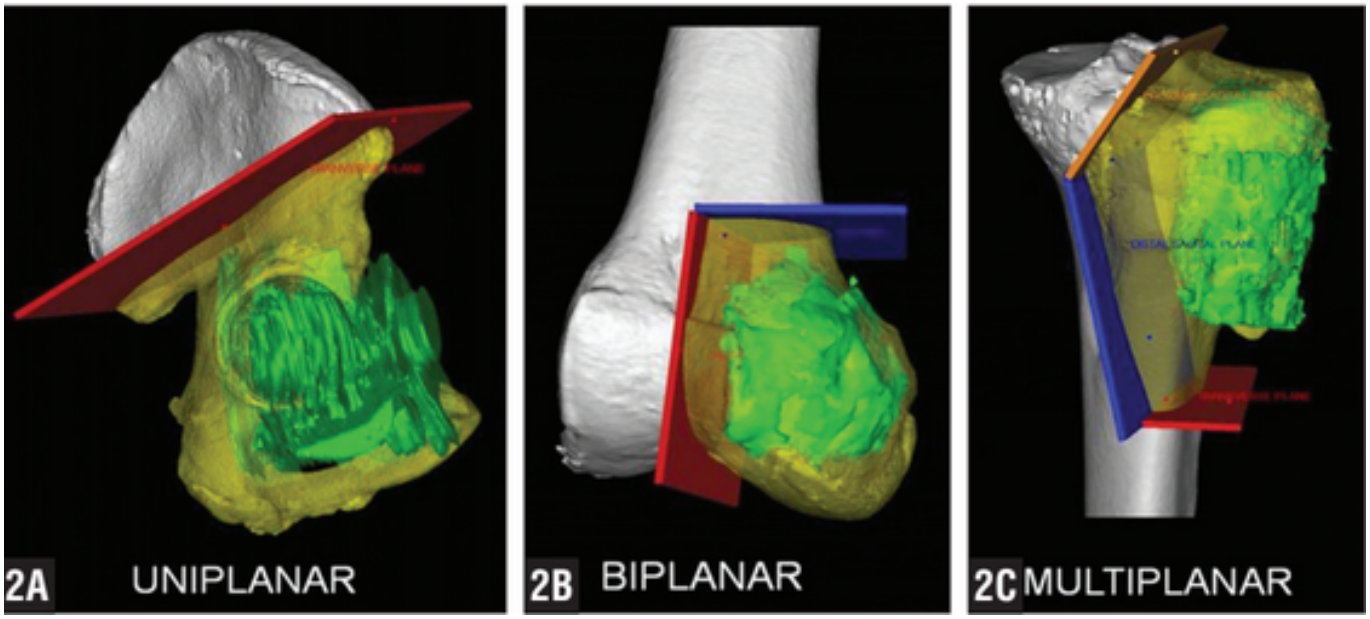


Figura 1: Distintos planos de corte al realizar una resección tumoral. Fuente: Bibliografía [4]

1.2. Planeamiento Virtual Preoperatorio

Durante la etapa de VPP se adquieren imágenes del paciente, las cuales serán procesadas para luego crear un **escenario virtual** interactivo donde estas imágenes reconstruyan un **modelo 3-D** de un órgano o región anatómica de la persona a operar (figura 2, página 5). Este escenario servirá para que el cirujano o equipo de cirugía pueda medir distancias oncológicas, planear y simular osteotomías de manera previa a la intervención quirúrgica.

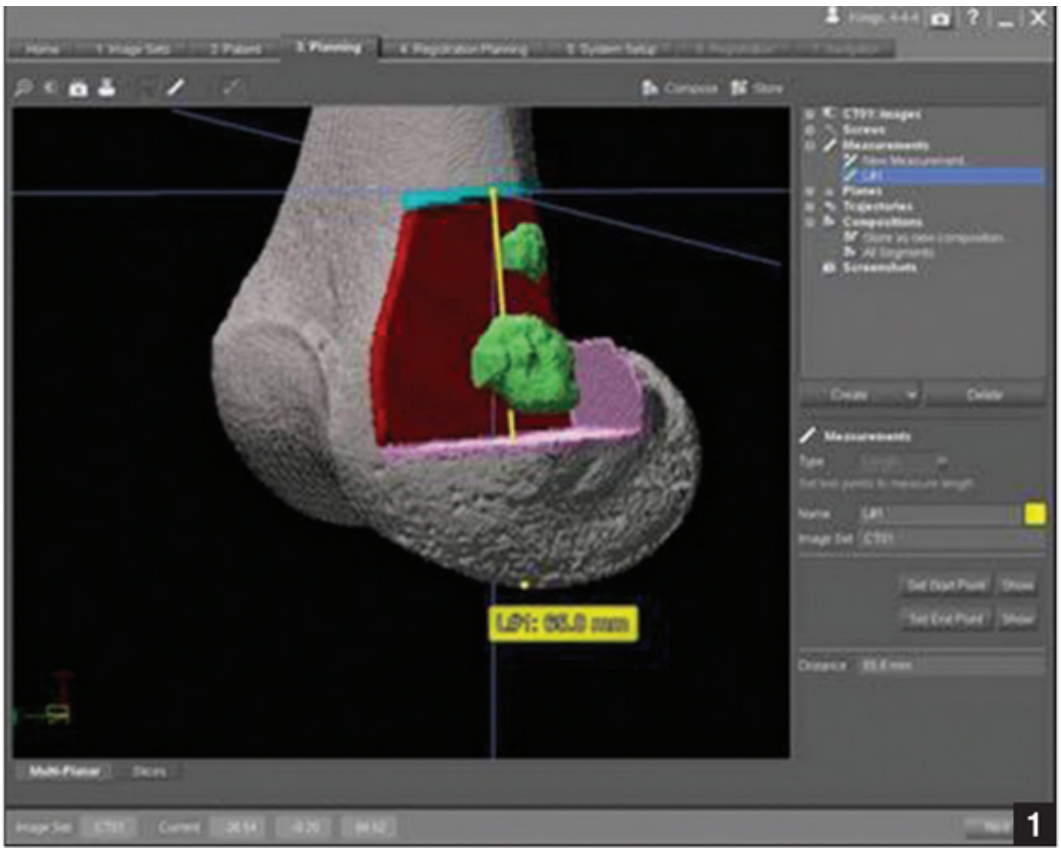


Figura 2: Planificación virtual de un paciente con un condrosarcoma (en verde). Se pueden observar los planos de corte en referencia con el tumor y el hueso a tratar. Fuente: Bibliografía [5]

1.3. Navegación Virtual Intraoperatoria

IVN es un procedimiento que permite la ejecución del VPP con un margen de error en el orden del milímetro [2]. Dicho error se obtiene de la cámara de navegación, al medir las imágenes que contienen los puntos fiduciarios y triangular su posición.

Durante la navegación intraoperatoria se utiliza un dispositivo denominado **"Navegador Quirúrgico"**. El mismo consta de una consola y una pantalla acoplados a un sistema óptico, que a su vez consta de un marcador láser. Este sistema genera una correspondencia entre la posición del puntero o marcador que se encuentra apuntando al paciente durante la intervención, con la consola que contiene un escenario virtual con las imágenes adquiridas (mediante Tomografía computada , Resonancia Magnética, etc) o con reconstrucciones 3-D (es decir contiene la planificación preoperatoria). Luego de un proceso de registración o alineamiento [2], se establece una correspondencia entre las coordenadas "reales" (corresponden al sistema de coordenadas relativas a la cámara) y el sistema de coordenadas virtual (correspondiente a el escenario virtual). Luego de una correspondencia de al menos 3 puntos, la posición del puntero podrá verse en la pantalla del sistema de navegación (figura 3, página 6)

Es la precisión de este sistema de navegación lo que lleva a que el error entre la planificación preoperatoria y el resultado de la intervención sea mínimo, como ya se mencionó anteriormente.

Existen dos medidas para el error de registración [32]:

- **Fiducial registration error (FRE):** Este tipo de error describe la correspondencia entre, un punto definido en la imágen y su mappeo al sistema de coordenadas del paciente. Es decir, el valor busca medir que tan buen "match" hay entre, el sistema de coordenadas de las imágenes generadas y el sistema de coordenadas reales del paciente. Este error depende del sistema de navegación utilizado al operar y se utiliza como medida de calidad de la registración entre las coordenadas reales y virtuales. El FRE se calcula como el valor root mean square (RMS) sobre todos los fiduciarios involucrados en la registración.
- **Target registration error (TRE):** Este tipo de error mide la distancia entre un punto expresado en las coordenadas virtuales y el mismo punto dado en coordenadas del paciente, pero transformado con una matriz de registración dada.

El TRE es el tipo de error mas significativo al describir la calidad de la registración, no obstante, es el mas difícil de medir. Esto se debe a que su valor depende de la distribución espacial de los puntos fiduciarios y, a su vez, de la relación entre los instrumentos del navegador y dichos marcadores. Analizar el TRE implica correlacionar puntos (distintos a los fiduciarios), tanto en la imágen como en el espacio del paciente, con gran precisión. Esto se logra en experimentos, llevados a cabo con marcadores implantados distribuidos uniformemente en superficies óseas de prueba. Dichas superficies se utilizan como puntos de medición, lográndose así una estimación del TRE.

Figura 3: Fotografía de un cirujano operando con un sistema de navegación. Fuente: Bibliografía [5]

2. Estado del Arte y futuro de las tecnologías CAS

El avance en tecnología CAS está vinculado principalmente con 4 campos [1] (figura 4, página 7):

1. El avance en tecnología de adquisición y procesamiento de imágenes.
2. El avance en el desarrollo de micro-instrumentos y micro-sensores.
3. Desarrollo de sistemas automáticos o robots en cirugía.
4. Avances en Realidad Virtual y en mayor medida Realidad Aumentada.

De todas estas áreas, son las dos últimas las que vieron el mayor avance en los últimos años, evidenciado por el desarrollo de los lentes de realidad aumentada y por los avances en cirugía robótica.

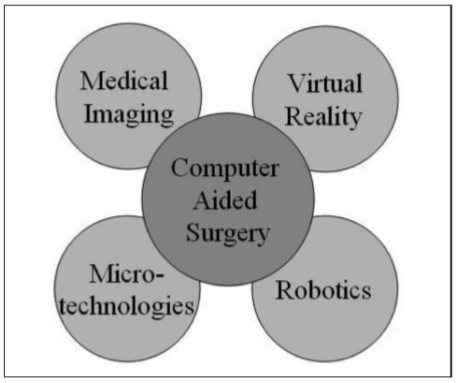


Figura 4: Principales áreas que avanzan con la tecnología CAS. Fuente: Bibliografía [1]

2.1. Cirugía Robótica.

En el contexto de la cirugía, los robots pueden ser brazos automáticos o sistemas telerobóticos. También se los puede clasificar como activos, semi-activos o pasivos [6]. Los de tipo semi-activos y pasivos transmiten los comandos y movimientos de un operador a los brazos del robot. Los de tipo activo por otro lado, tienen una rutina predefinida y funcionan mediante algoritmos de computadora, es decir, sin la necesidad de que un operador los manipule en tiempo real.

Los sistemas robóticos tienen ciertas ventajas sobre los humanos siendo la más importante, el aumento de la precisión durante la cirugía. Sin embargo, también tienen dos desventajas críticas: baja adaptabilidad (cada robot suele utilizarse para un tipo de cirugía en particular) y un bajo nivel de fuerza aplicada en los instrumentos manipulados por el sistema. Es por esto que estos sistemas se utilizan para pequeñas tareas y no para realizar cirugías completas.

Entre los sistemas de cirugía robótica más utilizados se encuentra el robot “Da Vinci” (Da Vinci Surgical System), creado por la empresa Intuitive Surgical Inc. (figura 5, página 8). Éste consiste de 1 a 2 consolas para cirujanos con un sistema de visualización 3D del escenario quirúrgico y un carro que contiene 3 o 4 brazos robóticos con sistemas de visualización endoscópicos y herramientas especiales. El sistema Da Vinci transmite los movimientos, que el cirujano realiza con sus manos, muñecas y dedos en la consola o estación, a los brazos robóticos del robot en tiempo real. Este sistema posee un mecanismo robusto que permite controlar y filtrar el temblor o pulso del cirujano, logrando disminuir el error en los movimientos ejecutados.

Figura 5: El robot Da Vinci de la compañía Intuitive Surgical. Fuente: Bibliografía [6]

2.2. Realidad Aumentada en cirugía

La cirugía asistida por realidad aumentada ("Augmented reality-assisted surgery" o ARAS en inglés) permite enriquecer la percepción sensorial, adicionando contenido virtual sobre la realidad. Esta tecnología funciona superponiendo una imágen generada por computadora sobre el campo de visión del cirujano. De esta manera se logra una vista compuesta sobre el paciente, mejorando la experiencia operatoria del cirujano.

ARAS puede ser utilizada para entrenar, preparar, simular o mejorar el resultado de una intervención. Esta tecnología se vale de diversos instrumentos. Los más comunes actualmente incluyen el uso de tabletas, proyectores o gafas de visión como, por ejemplo, los Google Glasses XE 22.1 o el modelo Vuzix Star 1200 XL. Estos dispositivos usan sistemas de proyección, rastreo y cámaras para generar un display del contenido virtual en el lente creando de manera efectiva, la ilusión de una realidad aumentada.

El tipo y cantidad de información que se impone depende de los requerimientos y procedimientos del equipo de cirujanos. AR ("Augmented Reality") es particularmente útil en visualizar estructuras críticas como por ejemplo vasos principales, nervios y otros tejidos vitales. Al proyectar estas estructuras directamente en el paciente, AR logra un aumento en la seguridad de la cirugía y el tiempo de ejecución de la misma [19]. Otra característica importante de los sistemas de AR, es su capacidad para controlar la opacidad de los objetos en escena, removiendo de esta manera distracciones en caso de emergencia.

Se espera que esta tecnología crezca en múltiples campos en los próximos años, aumentando su aplicación en cirugía [19].

Observar ejemplos del uso de esta tecnología en las figuras 6 y 7.

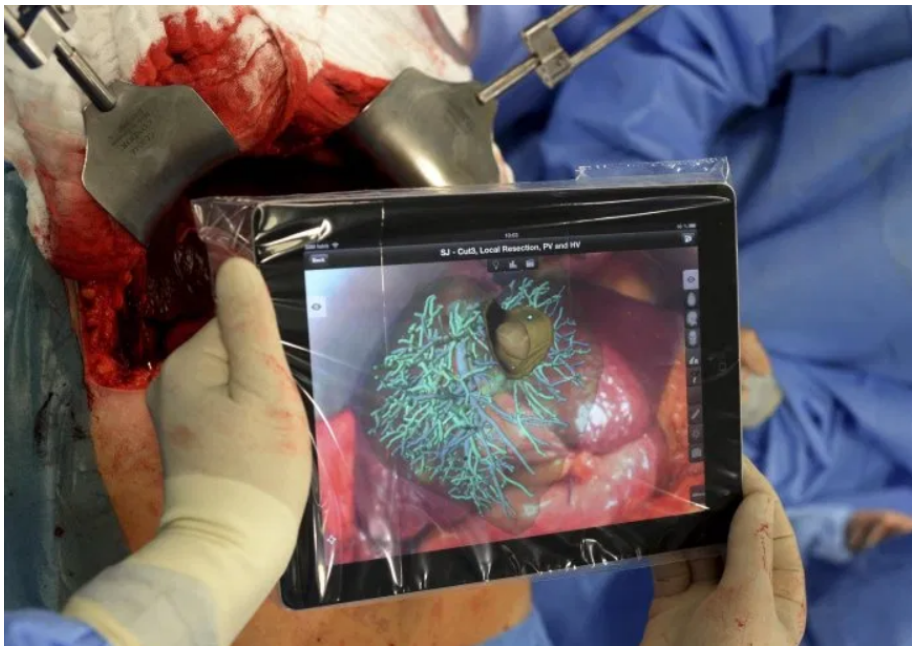


Figura 6: Uso de tecnología AR en una cirugía a corazón abierto. Fuente: Bibliografía [20]

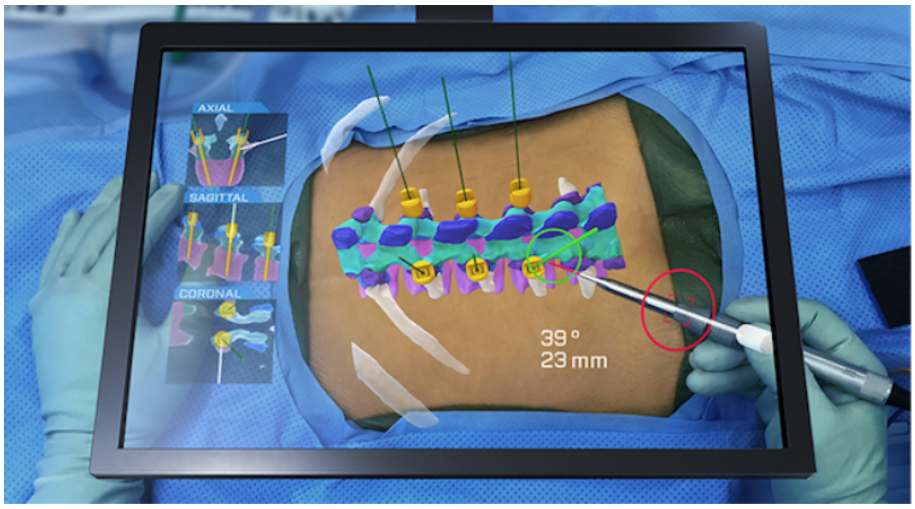


Figura 7: Uso de la tecnología AR en una operación de columna. Fuente: Bibliografía [21]

3. Motivación del Proyecto

Este proyecto propone ayudar de manera indirecta, en la reconstrucción de defectos óseos (posterior a una resección tumoral) en pacientes con tumores malignos, mediante el desarrollo de un software que permita seleccionar aloinjertos óseos. Diversos estudios muestran que, el uso de aloinjertos posteriormente a una resección da resultados aceptables, particularmente en reconstrucciones de tipo osteoarticular, transepifiseal e intercalar [25]. Desde 1950 la selección de aloinjertos óseos en base a su forma y tamaño se basaba en la comparación de imágenes adquiridas entre donante y paciente [27]. Esto presentaba errores por la escala de magnificación del instrumento de imágenes (Rayos X hasta ese entonces) y el hueso real, modificando la selección [28]. En 1970 la Tomografía Computada permitió refinar la selección, evitando imprecisiones gracias a la generación de cortes bidimensionales milimétricos [29]. Las últimas décadas introdujeron el uso de escenarios virtuales para planeamiento preoperatorio en donde, modelos anatómicos 3-D específicos para cada paciente pueden ser

reconstruidos a partir de imágenes médicas.

La determinación del tamaño y forma del injerto es un aspecto crítico, [26] debido a que esto asegura [25] estabilidad en articulaciones, buen sanado de heridas y evita problemas degenerativos en la superficie articular en aloinjertos osteoarticulares.

La determinación de dichas propiedades lleva tiempo y se realiza mediante mediciones manuales sobre los especímenes de un banco de datos o sobre modelos virtuales [26]. Este proyecto propone una herramienta de selección semiautomática (se requiere un ajuste fino para lograr resultados adecuados) basada en alineación computacional de formas, para seleccionar el espécimen que mejor se ajusta a la anatomía del paciente.

Poder construir una herramienta totalmente automática conlleva más tiempo y está fuera del alcance de este trabajo. Si bien es posible que en ciertos casos la herramienta aquí presentada funcione de manera automática, es recomendado usarla junto con un pequeño ajuste fino previo a la alineación. Esto se debe a que, un método de registración por nube de puntos tiene problemas en encontrar un mínimo global en su error cuando la situación inicial es desfavorable. En particular, esto sucede cuando se trata de ejecutar una rotación en un ángulo considerable debido a que el paso inicial en la rotación es pequeño y, en caso de ser mayor, éste provocaría mínimos locales que no llevarían a una convergencia global.

Es posible lograr una registración automática robusta si se implementa un método de búsqueda de correspondencia (el algoritmo Iterative Closest Point estándar no asume correspondencia entre puntos en las formas a alinear) mediante "Feature Detection and Extraction". Ciertos trabajos [30][31] muestran la posibilidad de alinear dos figuras en 3-D mediante correspondencia local entre grupos de vóxeles con características similares. Ciertas regiones de los objetos a alinear poseen características similares en forma, textura, curvatura, colorimetría entre otras. Dado un kernel de tamaño adecuado que contemple una mínima unidad de volumen o conjunto de vóxeles, es posible determinar en ambos volúmenes una correspondencia mediante una similitud en los valores de dichas características, estimadas computacionalmente.

Es posible, por último, intentar utilizar métodos de registración más sofisticados como aquellos implementados en base a "Level Sets" o "Elementos Finitos", sin embargo, estos tienen desventajas. Registrar múltiples objetos como es el caso de este trabajo sería imposible a una velocidad aceptable para los tiempos que dispone un cirujano. Por otro lado, los recursos utilizados son computacionalmente caros y requieren hardware avanzado. Considerando todo lo mencionado, se decidió proceder con el método de nube de puntos, el cual será explicado en la sección número 8 de este informe.

4. Especificaciones del Proyecto

En los siguientes incisos se detallan los objetivos que se proponen para este proyecto.

4.1. Objetivos de Mínima

A continuación, se enumeran las metas a alcanzar al finalizar el desarrollo del proyecto:

1. El sistema virtual desarrollado debe lograr una visualización e interacción adecuada de las geometrías que intervienen en el escenario virtual.
2. Capacidad de generar cortes bidimensionales mediante planos de cortes axiales, sagitales y coronales en los volúmenes.
3. Se deberá lograr obtener una cierta métrica que describa el nivel de ajuste de los volúmenes a la anatomía del paciente, para así, discriminarlo o no como un potencial injerto oseo.

4.2. Objetivos de Máxima

Si el tiempo disponible lo permite, se ampliará el alcance del proyecto con los siguientes objetivos:

1. Permitir ajuste manual de la registración obtenida.
2. Colorimetría de similitud entre piezas.
3. Aceleración del proceso mediante múltiples registraciones en simultáneo.
4. Poder ver en simultáneo las registraciones de varios huesos para compararlas, ya sea en el escenario de visualización o exportando los resultados.

5. Herramientas

En los siguientes incisos se detallan las herramientas utilizadas durante el proyecto. Éstas, forman parte de las especificaciones técnicas del mismo.

5.1. Insight Toolkit

ITK (en inglés "Insight Segmentation and Registration Toolkit") es una librería de software Open Source que sirve como Framework para desarrollar software orientado a procesamiento de imágenes (en particular segmentación y registración). La segmentación es el proceso de identificación y clasificación de datos a partir de una muestra digital de una imágen o volumen (en este proyecto se trata con volúmenes). Usualmente, esa imágen se obtiene por instrumentación como el resonador (MRI) o el tomógrafo. Por otro lado, la registración es un proceso en el que alineamos o buscamos correspondencia entre imágenes digitales. Por ejemplo, es posible alinear un corte de MRI con un scan de TC para aumentar la información al mostrar la combinación de ambos.

ITK nos provee de herramientas, algoritmos de segmentación, registración, para dos y tres dimensiones. También podemos encontrar algoritmos de mejoramiento y filtrado de imágenes ("Enhancement" en inglés) pero en menor cantidad.

ITK utiliza CMake (herramienta que será explicada en la sección 5 de este informe) para la configuración del proceso de construcción o "build" de los programas [10]. Esta librería se encuentra disponible en C++ y python.

En los siguientes dos incisos se introducirán los conceptos de Registración y Segmentación de imágenes digitales, los que serán mencionados reiteradas veces en este proyecto.

5.1.1. Proceso de Registración de imágenes digitales

Registración es el proceso de transformar imágenes, que se encuentran en distintos sistemas de coordenadas, a uno común. Esta técnica se utiliza en visión computacional, procesamiento de imágenes, reconocimiento de objetivos en sistemas militares, análisis de imágenes satelitales o biológicas, entre otras aplicaciones.

El proceso de registración es necesario para poder comparar o integrar las imágenes (o datos) obtenidas en distintas mediciones o muestreos. Por ejemplo, en el campo de las imágenes médicas tenemos muestras del cuerpo humano en distintos cortes (o slices en inglés) o tomados con distintos dispositivos (MRI o TC).

Los algoritmos de registración o alineamiento (extendido al caso tridimensional) se pueden clasificar en dos grandes categorías [11] :

1. Según la intensidad de los píxeles en la señal o **"Intensity based methods"**.
2. Según las características topológicas de la señal o **"Feature based methods"**.

En ambos casos siempre tendremos, por un lado una imágen a la cual llamaremos "móvil o template" y, por otro lado, una imágen a la cual llamaremos "fija o target". Este proceso conlleva la transformación espacial del objeto móvil para alinearlo con la imágen target. El marco de referencia de la imágen target es estacionario, en el momento en que las demás imágenes son transformadas para alinearse con ésta.

Los métodos basados en intensidad comparan patrones en la imágen mediante diversas métricas de correlación. El objetivo es encontrar alguna correspondencia. Por otro lado, los métodos basados en características encuentran correspondencia entre features en ambas imágenes como, por ejemplo, puntos, líneas, contornos o texturas. Al determinar esta correspondencia entre ambas imágenes, se calcula una transformación geométrica que realizará un mapeo de las imágenes móviles al sistema de coordenadas de la imágen fija.

Los algoritmos de alineación también se pueden clasificar en dos grupos, según el tipo de transformación aplicada [11]:

1. Transformación espacial **rígida o inelástica**.
2. Transformación espacial **deformable, no rígida o elástica**.

El primer grupo incluye operaciones básicas como por ejemplo la roto-traslación, pero sin deformar las escalas en la imágen. La segunda categoría incluye transformaciones que son capaces de deformar ("Warping" [11]) la geometría local de la imágen móvil para alinearla con la imágen de referencia.

De estos dos modelos de transformación, este trabajo solo utiliza las operaciones rígidas ya que es necesario mantener la forma original de los objetos a registrar. Por otro lado, las registraciones de tipo deformable suelen modelarse con ecuaciones diferenciales, en consecuencia, suelen ser más costosas computacionalmente llevando más tiempo en procesarse.

Las transformaciones se describen mediante parámetros, donde cada modelo dicta la cantidad de parámetros a utilizar. Estos parámetros forman un vector o una matriz que caracteriza la transformación global. Existen modelos paramétricos y modelos no paramétricos, los cuales transforman cada uno de los elementos de la imágen de manera arbitraria.

Las siguientes figuras muestran ejemplos de registración en 2D y 3D para casos deformables y rígidos:

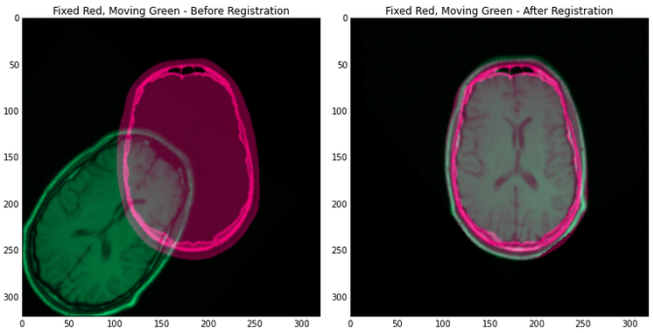


Figura 8: Registración en 2D de tipo rígida. Fuente: Bibliografía [12] .

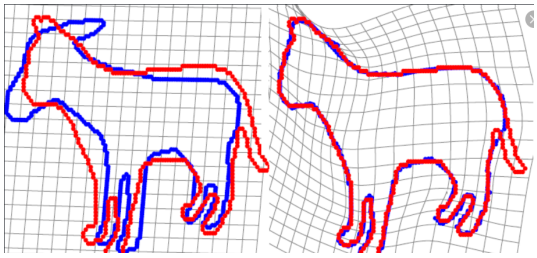


Figura 9: Registración en 2D de tipo deformable. Fuente: Bibliografía [13] .

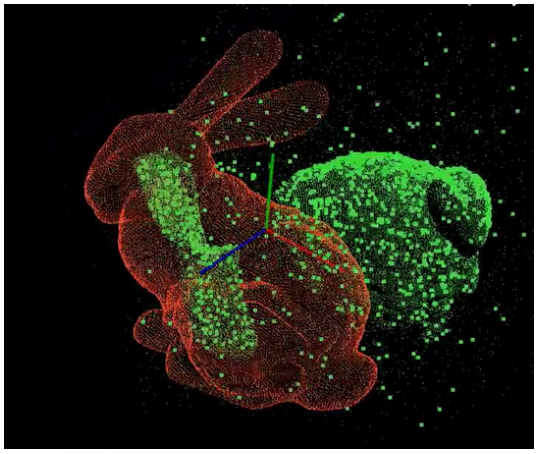


Figura 10: Registración en 3D de tipo rígida. Fuente: Bibliografía [14] .

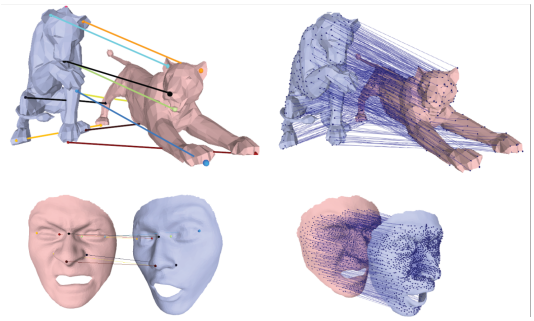


Figura 11: Registración en 3D de tipo deformable. Fuente: Bibliografía [15] .

5.1.2. Proceso de Segmentación de imágenes digitales

En procesamiento de imágenes y visión computacional, **segmentación** de imágenes es el proceso de partición de una imágen digital en distintos segmentos (conjunto de píxeles) que compartan características en común. El objetivo de esto es lograr simplificar la representación de una imágen en algo más fácil de analizar, o con más significado. A menudo, la segmentación es usada para detectar objetos o fronteras en imágenes.

En términos técnicos, la segmentación de imágenes es el proceso por el cual se le asigna una etiqueta con un valor a cada píxel, de modo que, píxeles con la misma etiqueta comparten ciertas características. El resultado de la segmentación es una serie de segmentos que en conjunto cubren toda la imágen original. Cada uno de los píxeles en una región son similares con respecto a ciertas características como por ejemplo [11]:

- Color
- Intensidad
- Textura

Los algoritmos de segmentación, se calibran de manera que las regiones adyacentes presenten diferencias significativas con respecto a sus características.

Cuando se aplica a un conjunto de imágenes, como por ejemplo en el caso de las imágenes médicas, los contornos resultantes de la segmentación pueden ser utilizados para reconstruir en 3D una estructura anatómica. Esto es lo que genera los volúmenes que vemos en los sistemas de visualización quirúrgicos como el que se utilizará en este trabajo.

En las figuras 12 y 13, se muestran ejemplos de segmentaciones aplicadas en 2D y 3D.

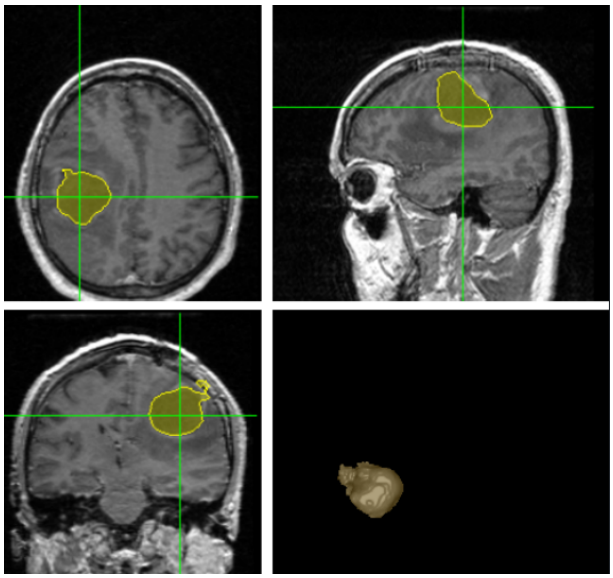


Figura 12: Segmentación 2D de tumor cerebral. Fuente: Bibliografía [16].

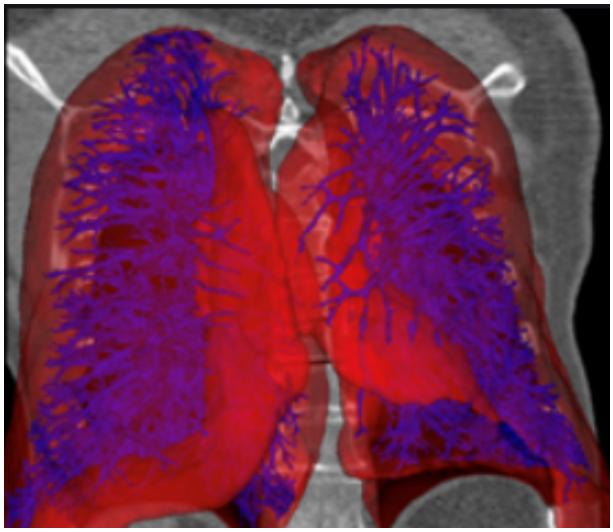


Figura 13: Segmentación 3D de árbol pulmonar. Fuente: Bibliografía [17].

5.2. Visualization Toolkit

VTK es una librería desarrollada por Kitware y disponible en C++ y python. VTK provee soporte para una variedad de algoritmos de visualización incluyendo técnicas de visualización de volúmenes, texturas, vectores y de modelado de figuras como polígonos, mallas o contornos.

Esta librería, nos provee de un Framework de visualización con distintas Widgets interactivas en 2-D y 3-D. También nos permite su integración con otras herramientas gráficas como Qt. Es importante destacar que VTK es cross-platform y funciona en sistemas Unix, Windows y Mac [18].

El uso de VTK se puede describir en el siguiente Workflow [18]:

1. Leer o generar datos.
2. Filtrar dichos datos.
3. Realizar el rendering o "renderizar" los datos.

4. Interactuar con los datos.

Los primeros dos pasos pueden omitirse, ya que para este proyecto, dispondremos de datos en un formato determinado (volúmenes STL). En los siguientes incisos se presentan los principales componentes que forman un escenario virtual en VTK y que completan el workflow presentado anteriormente.

5.2.1. VTK Actor

En VTK, un actor (figura 14) es todo objeto (ya sea en 2-D o 3-D) que aparece en el escenario virtual. Los actores se crean a partir de datos (vértices, coordenadas, etc) mapeados hacia el escenario a través de un objeto de VTK denominado "Mapper".

Los actores poseen características [18] como por ejemplo:

- Propiedades superficiales: opacidad, color, translucidez, etc.
- Representación: superficie, nube de puntos o en estructura de alambre.
- Texturas.

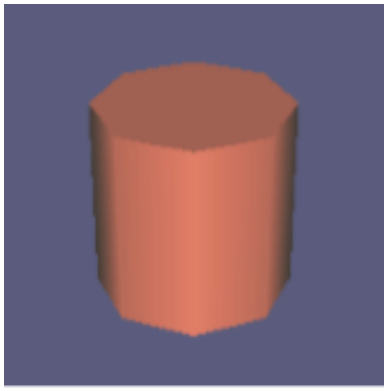


Figura 14: Ejemplo de un simple actor cilíndrico en una escena de VTK. Fuente: Bibliografía [18] .

En este trabajo los actores serán estructuras anatómicas (estructuras óseas en 3D y proyecciones de estos en distintos planos).

5.2.2. VTK Renderer

Rendering es el proceso que permite visualizar los resultados finales de un modelo o escena en 3D que no posee textura, sombreado o iluminación. Cuando "renderizamos" una escena o un modelo le agregamos (la computadora resuelve cálculos matemáticos) sombreado, reflexión, difusión, transparencia, refracción y luces. El motor de rendering de VTK permite visualizar la escena. Los objetos que conforman el escenario, es decir actores, cámara y luces, son contemplados por el motor de rendering de VTK.

5.2.3. VTK Render Window

La ventana de visualización de VTK contiene los actores que conviven en la escena virtual. La ventana posee propiedades como, por ejemplo, actores, colección de renderers, parámetros de iluminación, una cámara que da un foco de observación sobre la escena, entre otros. Esto puede observarse en la figura número 15.

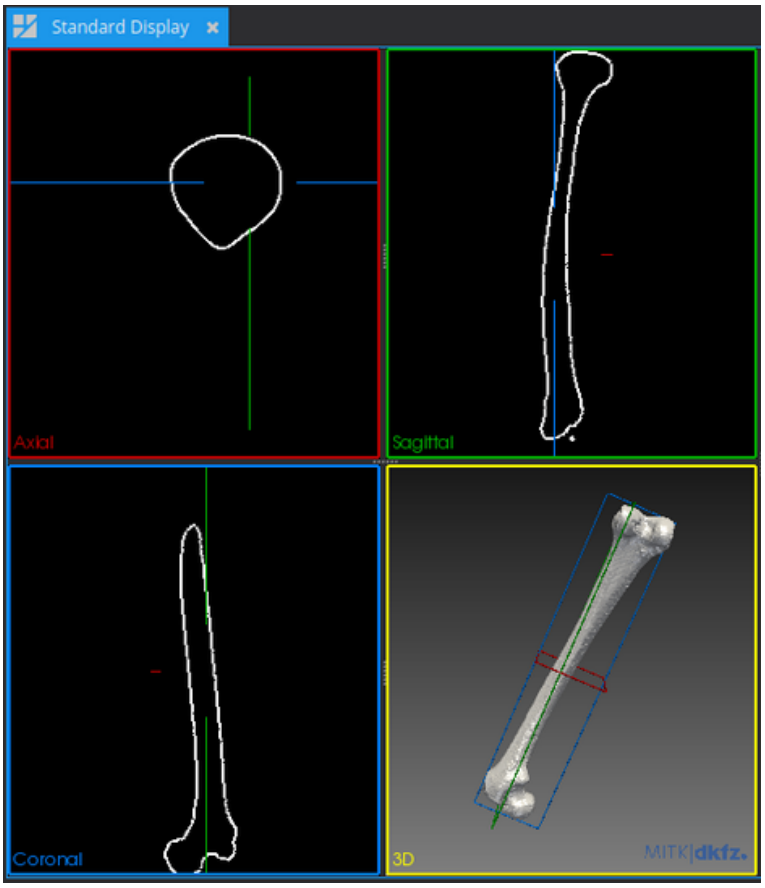


Figura 15: Layout de las ventanas de visualización del proyecto.

5.2.4. VTK Render Window Interactor

Cada usuario de VTK puede decidir cómo desea interactuar con los actores en la escena. Hay dos maneras distintas de controlar las interacciones en VTK [18]:

1. Generar una subclase de los estilos de interacción predeterminados de VTK.
2. Definir observadores que ejecuten un monitoreo continuo de eventos sobre el interactor y definir callbacks (o "commands") propios para implementar un estilo de interacción.

Diferentes estilos observan distintos eventos y realizan diferentes acciones en respuesta a dichos eventos, típicamente, modificando la posición de los actores (rotación, zoom, traslación) o del ángulo de la cámara en la escena. En este trabajo se utilizan ambas maneras de controlar las interacciones, como será explicado en la sección 8 de este informe.

5.3. Lenguaje de programación

El proyecto fue desarrollado en C++ por poseer dominio sobre este lenguaje. Otros aspectos a considerar fueron su performance, alta portabilidad, utilizable en múltiples dispositivos y con orientación en uso de objetos. Por otro lado las librerías utilizadas están documentadas, en su mayoría, en C++.

5.4. Qt

Para el entorno de desarrollo se seleccionó Qt. Éste se especializa en el diseño de interfaces gráficas. Además de las múltiples "Widgets" (elementos de la interfaz gráfica) que Qt provee para generar interfaces, también es particularmente útil la conexión "Signal-Slot" para ejecutar ciertas funciones importantes en el programa.

5.4.1. Mecanismo Signal-Slot

En Qt, los **Signals y Slots** son usados para la comunicación entre objetos. Este mecanismo de conexión entre ambos es una característica central de Qt que lo distingue de otros Frameworks. En la creación de interfaces gráficas, cuando una Widget (botón, etiqueta o miembro de la interfaz en general) se modifica, esperamos que las demás Widgets se vean afectadas. Generalizando, se busca que los objetos de cualquier clase en la interfaz se puedan comunicar mutuamente. Por ejemplo, si un usuario aprieta un botón de cerrado, esperamos que la función close() se ejecute.

Otros Toolkits logran este mecanismo de comunicación utilizando **"Callbacks"**. Un Callback es un puntero a una función, de manera que si deseamos que una función nos notifique sobre un evento, le debemos pasar un puntero a otra función (el Callback). Luego, la función que monitorea el evento puede invocar al Callback cuando lo considere apropiado. Si bien es común usar Callbacks, estos pueden ser poco intuitivos y puede haber problemas con el tipo de argumentos que reciben [7]. En la figura 16 puede observarse el esquema de funcionamiento de la metodología Signal-Slot.

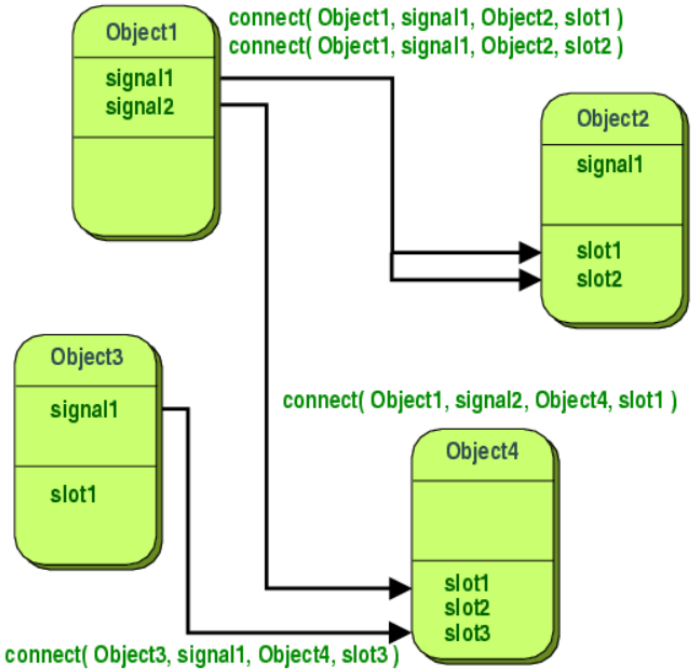


Figura 16: Esquema básico del mecanismo signal-slot de Qt Creator. Fuente: Bibliografía [7]

En Qt existe una alternativa al uso de los Callbacks: el mecanismo Signal-Slot. Una Signal es emitida cuando un evento en particular ocurre. Las Widgets de Qt tienen múltiples signals predefinidas, pero aun así también se pueden generar subclases de las Widgets y agregarles nuestras propias signals. Un Slot es una función que es llamada en respuesta a una signal en particular. Las Widgets de Qt tienen múltiples slots predefinidos y nuevamente es posible generar subclases de las Widgets y agregarles nuestros propios slots para manipular las signals de interés.

Es importante mencionar que las signals y slots están desacoplados: una clase puede emitir señales y luego distintos slots pueden o no suscribirse a estas señales. El mecanismo de signal-slot de Qt asegura que, si se conecta una signal a un slot, el slot va a ser llamado con los parámetros de la signal en el momento correcto. Las signals y slots pueden tomar argumentos de cualquier tipo. Se dice que son "type safe" [7].

Todas las clases que hereden de **QObject** (clase base de Qt) o cualquiera de sus subclases (QWidget por ejemplo) pueden contener signals y slots. Las signals son emitidas por objetos que cambian su estado tal que esto afecte otro objeto. En este mecanismo, no es necesario que el objeto sepa si algo está recibiendo la señal que éste emite. Los slots pueden ser usados para recibir signals, pero también se pueden comportar como funciones de la clase (member functions) del objeto receptor de manera normal. Tal como un objeto no es notificado

necesariamente si algo recibió su señal, un slot tampoco es notificado si tiene señales vinculadas a él. Esto asegura que los componentes creados utilizando Qt sean independientes.

Es posible conectar tantas signals como se quiera a un solo slot y una signal puede ser conectada con tantos slots como se requiera. También es posible lograr una conexión signal-signal (se emite la segunda luego de la primera). Este mecanismo es una herramienta poderosa y será utilizada frecuentemente en este trabajo.

5.4.2. Signals

Las **Signals** son funciones de acceso público ("Public") y pueden ser emitidas desde cualquier lado, no obstante los manuales de Qt aconsejan que éstas sean emitidas solo por la clase que define esta señal y a lo sumo subclases [8]. Cuando se emite una señal, los slots conectados a ésta generalmente se ejecutan de manera inmediata. Cuando esto sucede, el mecanismo signal-slot es totalmente independiente de cualquier evento que ocurra en el **"Event Loop"** de la GUI. La ejecución del código que continua a la emisión de la señal ocurrirá una vez que todos los slots terminen su ejecución y retornen. Si una señal tiene como objetivo varios slots, estos se ejecutaran cuando les llegue la señal de manera secuencial en el orden en el que fueron conectados. Es importante destacar que las señales no pueden tener un tipo de retorno, ya que su único objetivo es desencadenar el slot [8].

5.4.3. Slots

Un slot es invocado cuando una señal conectada a él es emitida. Los slots son funciones normales de C++ y pueden llamarse de manera normal; su única característica especial es que se le pueden conectar señales.

Ya que los slots son funciones de una clase, pueden ser llamados directamente. Sin embargo, cuando actúan como slots pueden ser llamadas mediante el mecanismo signal-slot, sin importar el nivel de acceso [7] dentro de la clase (protected, private, public). Esto significa que una señal emitida desde una instancia de una clase puede activar un slot de otra instancia. La instancia receptora ejecuta el slot aunque éste sea privado y sin importar si la clase se relaciona o no con la señal emisora.

5.4.4. Mecanismo Signal-Slot y Callbacks

Comparado con los callbacks, el mecanismo signal-slot es más lento, aunque provee más flexibilidad. En general, emitir una señal que está vinculada con un slot es aproximadamente 10 veces más lento que llamar a la función de manera directa [7]. Esto se debe a que es necesario localizar todos los slots posibles, verificar si hay otras señales en el camino y asegurarse que ningún receptor de la señal haya sido destruido durante su emisión. Dicho esto, la simplicidad y flexibilidad que brinda el mecanismo compensa esta demora, la cual puede que el usuario no note.

5.5. Smart Pointers

Durante este trabajo se hace uso extensivo de los Smart Pointers. Utilizar esta metodología permite optimizar el consumo de memoria y evitar "memory leaks" (principalmente en la interfaz gráfica que corre en el hilo principal). De esta manera, se logra que la memoria RAM no crezca descontroladamente durante el tiempo de ejecución o el tiempo en el que la interfaz esté abierta. Cuando creamos un objeto de VTK, a menudo lo utilizamos por un tiempo muy corto y cuando salimos del scope dicho objeto no se borra, sino que queda alojado en memoria. Cuando creamos un objeto en VTK usualmente lo hacemos mediante la función "New" de la clase del objeto:

```
vtkObject* Object_x = vtkObject::New();
```

Luego debemos eliminarlo manualmente para evitar memory leaks de la siguiente manera:

```
Object_x->Delete();
```

Lo anterior es un proceso tedioso, puesto que tendríamos que saber cuando eliminar cada objeto creado y evaluar si éste ya no nos será más útil. Para esto existe una clase de VTK llamada vtkSmartPointer, que gestiona este proceso de manera automática, mediante el uso de contenedores de referencia. La forma de crear un objeto con smart pointers es la siguiente:

```
vtkSmartPointer<vtkObject> Object_x = vtkSmartPointer<vtkObject>::New();
```

5.6. CMake

CMake es una familia de herramientas diseñada para construir, probar y empaquetar software. CMake se utiliza para controlar el proceso de compilación del software, mediante el uso de ficheros de configuración independientes de la plataforma que se esté utilizando.

CMake genera archivos denominados **"makefiles"** que pueden usarse en el entorno de desarrollo deseado. Es comparable al GNU build system de Unix en que el proceso es controlado por ficheros de configuración, en el caso de CMake llamados **"CMakeLists.txt"**. Al contrario que el GNU build system, que está restringido a plataformas Unix, CMake soporta la generación de ficheros para varios sistemas operativos. Esto elimina la necesidad de tener varios conjuntos de ficheros para cada plataforma [9] .

CMake puede trabajar con proyectos que requieren la creación de ejecutables, antes de generar código compilable para la aplicación final. Su diseño de código abierto y extensible permite que CMake se adapte según sea necesario para proyectos específicos. En concreto, CMake fue utilizado en este proyecto para poder "linkear" ITK y VTK, dando además la posibilidad de trabajar multiplataforma en Linux y Windows.

Por último, CMake puede generar archivos de proyecto para varios IDEs destacados, como Microsoft Visual Studio, Eclipse y Qt. En este caso se utilizo CMake en conjunto con Qt.

5.6.1. CMake y Qt

Qt detecta CMake según la variable "PATH" (dirección) que se especifica en los ficheros de configuración del proyecto.. Mediante un archivo de Cache, Qt corre CMake de manera automática para refrescar o actualizar la información de un proyecto, cada vez que se edita un archivo de configuración CMakeLists.txt vinculado al mismo.

5.7. Multithreading

La última herramienta que debe ser mencionada es el "Multithreading" (múltiples hilos de ejecución). Los procesadores (CPUs) con capacidad para Multithreading tienen soporte en hardware para ejecutar múltiples hilos de ejecución de un programa, donde cada núcleo o "core" del procesador puede ocuparse de una determinada cantidad de hilos.

Bajo este paradigma, un proceso se divide en múltiples procesos denominados "threads" o hilos. Cada uno de estos posee su propio camino de ejecución, lo que hace posible la comunicación entre hilos. Es importante destacar que, aunque cada hilo posee su propio registro y stack, es necesario tomar recaudos de que dos o más hilos separados no accedan a una misma posición de memoria.

Debido a que el programa desarrollado en este proyecto consta de varias tareas por separado, fue conveniente la implementación del Multithreading. De esta manera, podemos tratar las tareas independientes como hilos de ejecución separados.

Si bien los tiempos de ejecución de un solo hilo no son mejorados [8] , sino que por el contrario, degradados, esta técnica fue utilizada en este trabajo con dos objetivos:

1. Separar el GUI Loop del proceso de registración de imágenes.

2. Ejecutar múltiples registraciones en simultáneo.

De esta manera podemos continuar interactuando con la interfaz mientras se está llevando a cabo una o más registraciones en paralelo. En la figura 17 se puede observar una descripción sencilla del esquema propuesto.

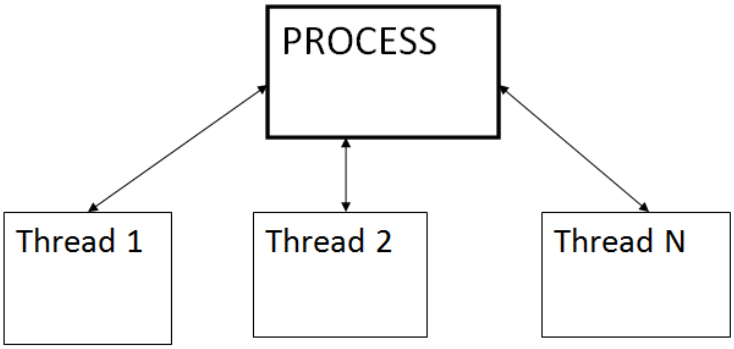


Figura 17: Esquema básico de un proceso separado en múltiples hilos. Fuente: Multiprogramming vs Multiprocessing vs Multitasking vs Multithreading, Java T Point.

6. Estructura del proyecto.

6.1. MITK

MITK es una librería para desarrollo de aplicaciones basadas en procesamiento de imágenes y visualización. Tal como VTK e ITK están basados en openGL, MITK está basado en VTK e ITK y requiere CMake para la construcción de sus proyectos.

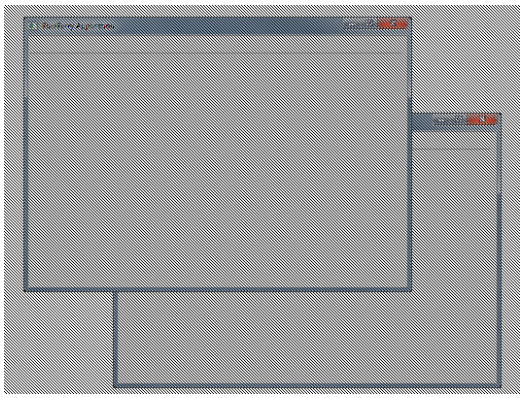
MITK posee un Framework propio llamado **Blueberry**, que permite la construcción de aplicaciones ejecutables que contengan interfaces gráficas. Dentro de MITK existe una aplicación denominada "MITK Workbench", que sirve como entorno principal de visualización.

La base de la Workbench está construida de manera similar a una interfaz gráfica de Eclipse. Está compuesta por los siguientes elementos (enumerados según jerarquía decreciente):

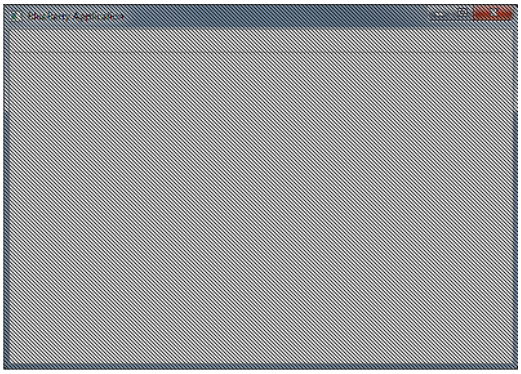
1. **Workbench**: esta es una colección de ventanas (figura número 18(a), página 21).
2. **WorkBench Window**: es una ventana perteneciente a la Workbench (figura número 18(b), página 21). Contiene elementos denominados páginas.
3. **Página**: la página o solapa (figura número 18(c), página 21) constituye la parte interior de la ventana, es decir, todo menos la barra superior que contiene el título de la ventana. La página puede contener barras de menú, de herramientas, barras de estado o perspectivas.
4. **Perspectivas**: éstas (figura número 18(d), página 21) son contenedores visuales para sets de vistas o editores, arreglados en un Layout determinado. Puede haber múltiples perspectivas dentro de una misma ventana, pero solo una es visible a la vez.
5. **Editor**: el editor (figura número 19(a), página 21) contiene documentos, imágenes o STLs con los que el usuario interactuará.
6. **Vistas**: por último las vistas (figura número 19(b), página 21) contienen las funcionalidades, por ejemplo el data manager o alguna interfaz desarrollada en un plugin.

La estructura modular de la Workench utilizada en este trabajo incluye:

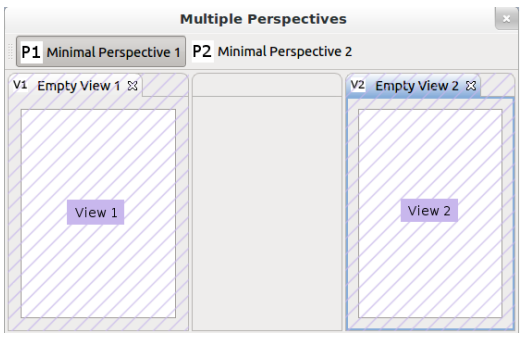
1. **Menú principal**: permite cargar y desechar datos.
2. **Display de 4 ventanas**: es el núcleo de la aplicación. Incluye una vista 3-D y tres vistas en 2-D. Aquí ocurre la interacción con los datos cargados.
3. **Data Manager**: esta sección del Workbench sirve para manejar toda la información y datos cargados en el menú.



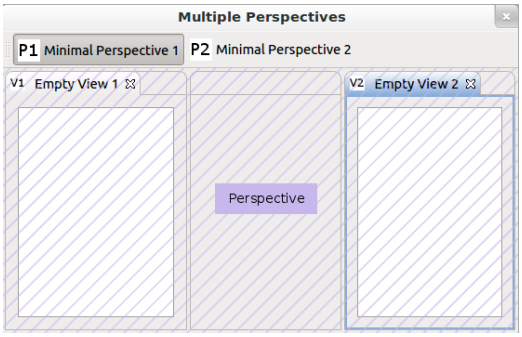
(a) Workbench genérico de MITK.



(b) Workbench Window genérica en MITK.

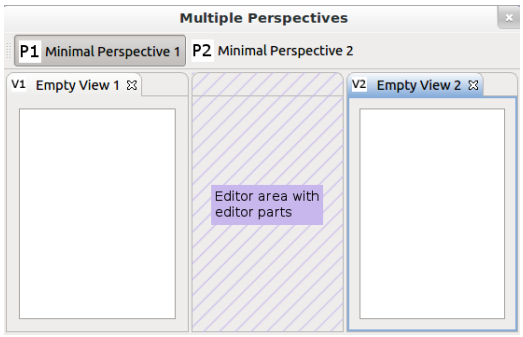


(c) Workbench Page genérica en MITK.

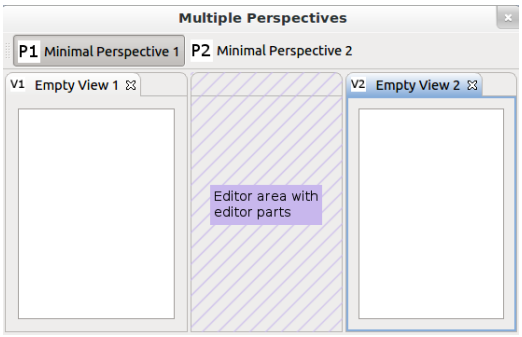


(d) Workbench Perspective genérica en MITK.

Figura 18: Fuente: Manual de MITK.



(a) Workbench Editor genérico en MITK.



(b) Workbench View genérica en MITK.

Figura 19: Fuente: Manual de MITK.

4. **Módulos activos**: muestra los módulos cargados junto con su funcionalidad y su interfaz propia.
5. **Plugins**: MITK contiene un sistema de desarrollo que permite extender la Workbench mediante la creación de Plugins. Estos tienen diversas funcionalidades según sea necesario, realizando funciones especificas que pueden, por ejemplo, manipular o procesar los datos visualizados en las ventanas.

6.1.1. Blueberry

Como ya se mencionó anteriormente, Blueberry es el Framework utilizado para generar aplicaciones de MITK. En este proyecto, se utilizó la Workbench de visualización. Para esto fue necesario configurar MITK para que funcione junto con BlueBerry.

6.1.2. CTK

Inicialmente, además de funcionar como generador de aplicaciones MITK, Blueberry servía como Framework para generar plugins de MITK. Sin embargo, esta funcionalidad fue removida en favor del "Common Toolkit" o CTK, el cual es un framework más robusto que Blueberry para generar plugins. Es decir que, para este proyecto se utiliza la creación de aplicaciones con Blueberry pero el manejo de los Plugins lo realiza CTK.

El objetivo de CTK es apoyar el desarrollo de lo que se denomina "Image Computing", por tanto, su Framework es open source. CTK trabaja en tópicos no cubiertos por otros toolkits como VTK, ITK o MITK mismo. Algunos tópicos cubiertos por CTK incluyen, el manejo de imágenes DICOM, Widgets y la aplicación utilizada en este caso: un Framework de Plugins.

6.2. Configuración del proyecto

El proyecto se configura mediante archivos de tipo CMakeLists y CMAKE files. Los archivos de configuración son extensos, por lo que este documento solo destaca los más importantes, es decir, aquellos usados para configurar dependencias externas, plugins, CTK y Blueberry.

6.2.1. Configuración del archivo CMakeLists.txt del proyecto

El primer fichero a configurar es el archivo CMakeLists principal. Es importante notar que existe un archivo denominado "SuperBuild" que permite configurar y generar el build de todas las dependencias de MITK de manera automática. Esta opción fue descartada, por lo que todas las dependencias de MITK están especificadas de manera explicita en el archivo CMakeLists.txt. La configuración se muestra en los siguientes pasos (se mostrarán solo los pasos más importantes).

Primero se debe verificar la versión de CMake mínima para que el proyecto funcione. Luego debemos especificar el nombre del proyecto y de la aplicación:

```
cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
set(MY_PROJECT_NAME template)
set(MY_APP_NAME templateApp)
```

Luego se debe especificar el tipo de build. Desde Qt, seleccionamos el tipo de build como RelWithDebInfo"(release with debug information). Sin embargo, debemos indicarle a CMake las distintas opciones de build disponibles:

```
set_property(CACHE CMAKE_BUILD_TYPE PROPERTY
             STRINGS "Debug" "Release" "MinSizeRel" "RelWithDebInfo")
```

Luego ordenamos a CMake que encuentre la versión de MITK especificada y verifique compatibilidad:

```
find_package(MITK 2016.11.0 REQUIRED)

if(COMMAND mitkFunctionCheckMitkCompatibility)
  mitkFunctionCheckMitkCompatibility(VERSIONS MITK_VERSION_PLUGIN_SYSTEM 1 REQUIRED)
else()
  message(SEND_ERROR "Su versi n de MITK es obsoleta")
endif()
```

Luego especificamos el path con dependencias externas necesarias para el projecto que CMake debe construir:

```
set(${PROJECT_NAME}_MODULES_PACKAGE_DEPENDS_DIR "${PROJECT_SOURCE_DIR}/CMake/PackageDepends")
list(APPEND MODULES_PACKAGE_DEPENDS_DIRS ${${PROJECT_NAME}_MODULES_PACKAGE_DEPENDS_DIR})
```

Debemos especificar la versión de Qt a utilizar:

```
if(MITK_USE_Qt5)
  set(QT_QMAKE_EXECUTABLE ${MITK_QMAKE_EXECUTABLE})
  add_definitions(-DQWT_DLL)
endif()
```

Para habilitar el sistema de Plugins de CTK debemos especificarle a CMake:

```
include(${CMAKE_CURRENT_SOURCE_DIR}/Plugins/Plugins.cmake)
ctkMacroSetupPlugins(${PROJECT_PLUGINS}
                     APPS ${_apps_fullpath}
                     BUILD_OPTION_PREFIX ${PROJECT_NAME}_
                     BUILD_ALL ${${PROJECT_NAME}_BUILD_ALL_PLUGINS})
```

6.2.2. Configuración del archivo MITK.cmake del proyecto

Este proyecto se vincula con MITK mediante la carpeta donde MITK está instalado. Sin embargo, el archivo MITK.cmake que se encuentra en la carpeta del proyecto realiza algunas verificaciones sobre MITK.

Lo primero que debemos hacer, es deshabilitar el SuperBuild. Luego se debe habilitar Blueberry, CTK y Qt. La opción "Build all MITK plugins" estará en OFF ya que en esta sección y en la octava, especificaremos los plugins que deseamos construir junto con el proyecto:

```
option(MITK_USE_SUPERBUILD "Use superbuild for MITK" OFF)
option(MITK_USE_BLUEBERRY "Build the BlueBerry platform in MITK" ON)
option(MITK_BUILD_ALL_PLUGINS "Build all MITK plugins" OFF)
option(MITK_USE_CTK "Use CTK in MITK" ${MITK_USE_BLUEBERRY})
option(MITK_USE_Qt5 "Use Qt 5 library in MITK" ON)
```

Por último, especificamos los directorios de ITK, VTK y Qt que MITK usará. Al instalar MITK, éste viene con una versión de ITK y VTK, sin embargo también es posible especificar un directorio propio de estas dos librerías en caso de que se las haya instalado previamente o se requiera otra versión:

```
set(my_itk_dir ${ITK_DIR})
set(my_vtk_dir ${VTK_DIR})
set(my_qmake_executable ${QT_QMAKE_EXECUTABLE})
```

6.2.3. Configuración de los archivos de la aplicación

En el archivo CMakeLists principal del projecto se definió el nombre del proyecto como "template" y la aplicación como "templateApp". En dicho archivo nos encargamos de configurar el proyecto. Ahora configuraremos el archivo CMakeLists propio de la aplicación. Lo único que haremos en este archivo será crear un subdirectorio para la aplicación dentro de la carpeta de proyectos.

```
if(${PROJECT_NAME}_Apps/${MY_APP_NAME} OR ${PROJECT_NAME}_BUILD_ALL_APPS)
  add_subdirectory(${MY_APP_NAME})
endif()
```

Luego configuraremos el archivo "Apps.cmake". Aquí le indicaremos a CMake que debe construir la aplicación "templateApp":

```
option(${PROJECT_NAME}_Apps/${MY_APP_NAME} "Build the ${MY_APP_NAME}" ON)
set(PROJECT_APPS
  Apps/${MY_APP_NAME}^^${PROJECT_NAME}_Apps/${MY_APP_NAME}
)
```

La primera opción en ON es importante, ya que CMake la toma desde el CMakeLists.txt principal del proyecto y esta opción alimenta a una función de CTK ("ctkFunctionSetupPlugins()"), que permite habilitar todas las dependencias que requieren los plugins de la aplicación en run-time.

Ahora configuraremos un nuevo CMakeLists.txt para definir la aplicación llamada “templateApp”. Además, excluiremos aquellos plugins que no deseamos construir junto con nuestro proyecto. Esto es necesario debido a que, si en run-time alguno de nuestros plugins está vinculado con otro propio de MITK, el proyecto fallará al construirse. La línea “org.mitk.gui.pluginName” representa un plugin determinado de MITK. Se debe especificar cada uno que se desee excluir. Un ejemplo posible sería: “org.mitk.gui.qt.photoacoustics.simulation”.

```
project(templateApp)
set(_exclude_plugins
org.mitk.gui.plugin_name
)
```

Luego, para crear la aplicación, la cual estará basada en el framework Blueberry, utilizamos la función para crear aplicaciones de Blueberry con los siguientes parámetros:

```
mitkFunctionCreateBlueBerryApplication(
    NAME ${MY_APP_NAME}
  DESCRIPTION "MITK - ${MY_APP_NAME} Application"
  EXCLUDE_PLUGINS ${_exclude_plugins}
  ${_app_options}
```

Continuando, vincularemos algunos módulos con la aplicación. El más importante es el que contiene Widgets de Qt, ya que éste fue utilizado en funciones del plugin:

```
mitk_use_modules(TARGET ${MY_APP_NAME}
  MODULES MitkAppUtil
  PACKAGES Qt5|Widgets
)
```

Por último, configuraremos un archivo denominado “target libraries” en donde habilitaremos los plugins para esta aplicación. Las primeras dos líneas permite que la aplicación utilice plugins del resto del proyecto. La última línea habilita el plugin “Data Manager”, el cual será utilizado como una View en la Workbench.

```
set(target_libraries

#Habilito plugin del proyecto para esta aplicaci n.
  template_templateApp

#Habilito plugin externo: Data Manager.
  org_mitk_gui_qt_datamanager
```

6.2.4. Configuración de plugin

Para activar un plugin se le debe agregar a un archivo de CMake. En este caso el archivo se denominó “Plugins.cmake”. Para agregar un plugin al proyecto se debe especificar el nombre del proyecto, el del plugin, y activarlo con la flag “ON”. Véase a continuación:

```
set(PROJECT_PLUGINS
  Plugins/template.templateApp:ON
)
```

7. Workbench

7.1. Data manager

Todos los datos importados en la aplicación figuran como un objeto de MITK denominado “nodo”. Estos se guardan en una estructura que se puede considerar un contenedor, denominado “Data Manager” (figura 20), el cual se puede ver a la izquierda de la aplicación. Los nodos guardan información sobre el nombre del dato importado, su color, tipo de representación, vértices, polígonos, entre otros campos. Durante los procesos ejecutados se cargarán datos desde y hacia el Data Manager, con el objetivo de procesar o ingresar outputs de los métodos que requieran acceso al contenedor.

Figura 20: Contenedor de nodos o "Data Manager".

7.2. Interfaz principal

La vista principal, ubicada a la derecha de la aplicación, contiene los controles principales para ejecutar, configurar e inicializar el procedimiento de registración y procesos asociados. Esta interfaz se divide en 4 secciones (ver figura 21):

1. **Panel de inicialización y carga de datos**: este panel permite inicializar los nodos presentes en el Data Manager y luego seleccionar el nodo target y el/los nodos templates.

2. **Panel de prealineación manual**: permite alinear manualmente superficies mediante una configuración de escalas y una rotación anclada en el origen de coordenadas o en el centro de cada objeto.

3. **Panel de ejecución**: contiene herramientas para la ejecución del alineamiento, la configuración del escenario y el cálculo de errores, junto con la generación de un mapa de color según proximidad.

4. **Panel de configuración de parámetros**: contiene los campos de input para modificar hiperparámetros de la alineación.

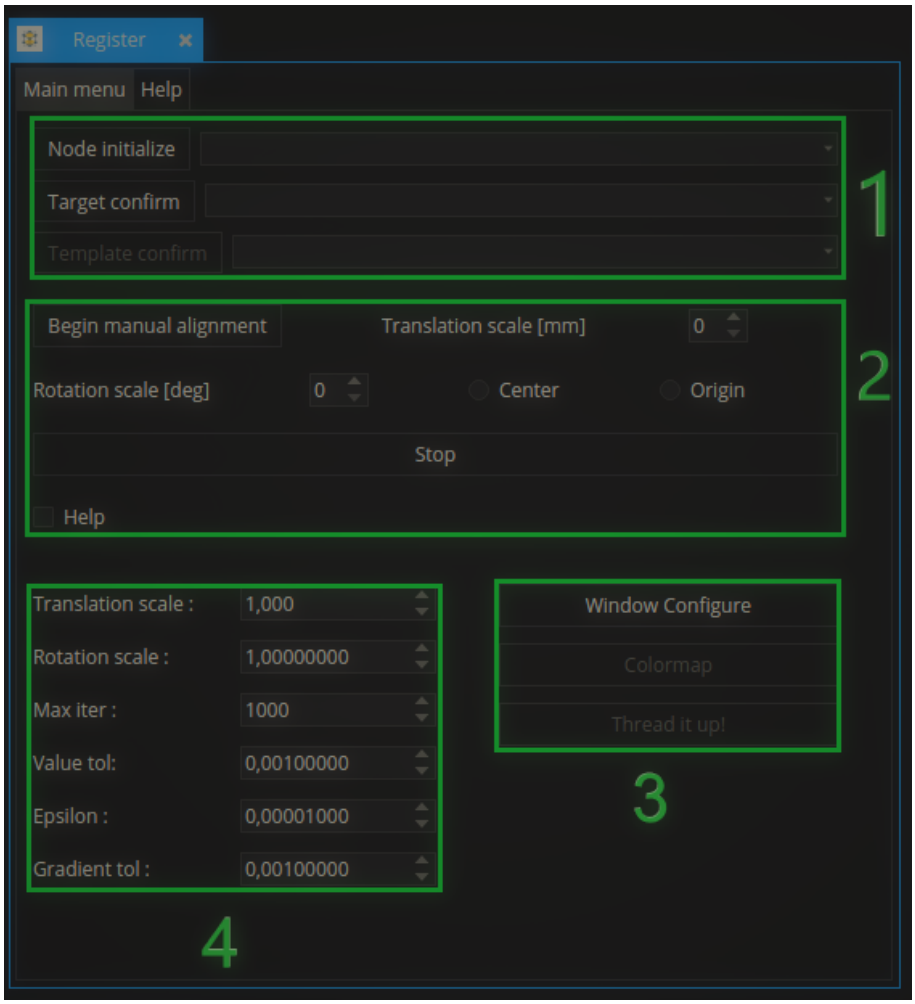


Figura 21: Aspecto de la interfaz de trabajo.

7.3. Layout de visualización.

La interfaz de visualización se encuentra adyacente a la de control y configuración. La misma consta de Render Windows de MITK donde se pueden observar y realizar modificaciones sobre los objetos importados en el Data Manager.

7.3.1. Ventana Principal

Esta ventana (figura 22) permite observar todos los objetos importados en el Data Manager y habilitados en la visualización. Posee tres planos virtuales para desplazarse sobre los objetos y obtener cortes bidimensionales a partir de estos.

A su vez, los objetos de esta ventana tienen propiedades configurables como su color y estructura. En esta ventana en particular, existen tres tipos de representaciones posibles de los objetos:

- **Representación Surface**: consta de una representación de la superficie del objeto resultando de la interpolaciones de los datos que contiene el PolyData. Puede observarse en la figura 23 (a).
- **Representación Wireframe**: consta de una representación del PolyData con una estructura de alambre. Puede observarse en la figura 23 (b).
- **Representación Point Cloud**: consta de una representación del PolyData como una nube de puntos. Puede observarse en la figura 23 (c).

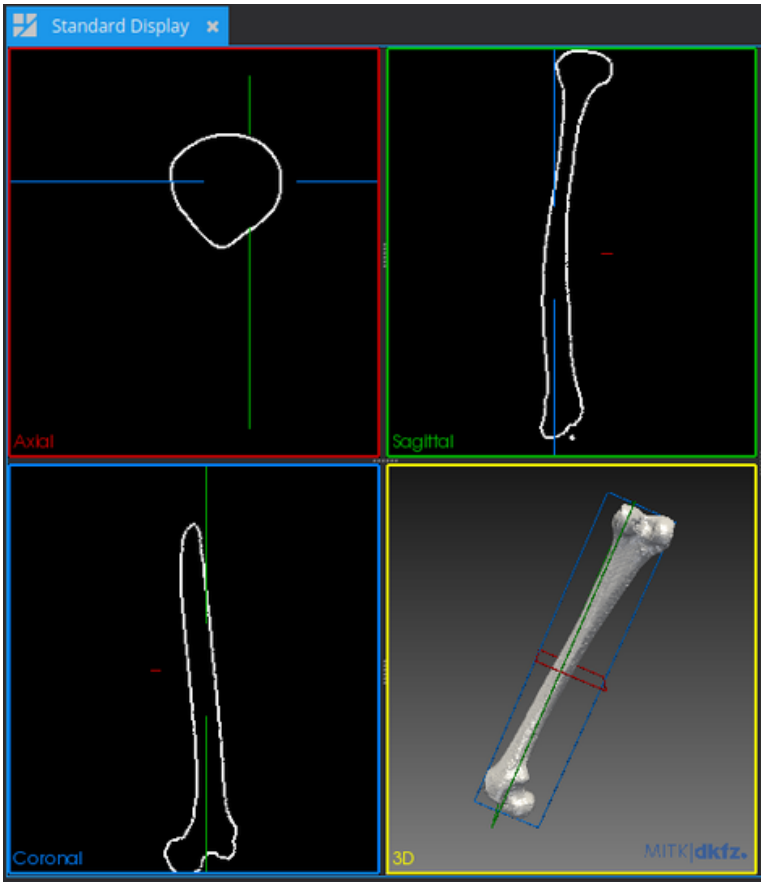
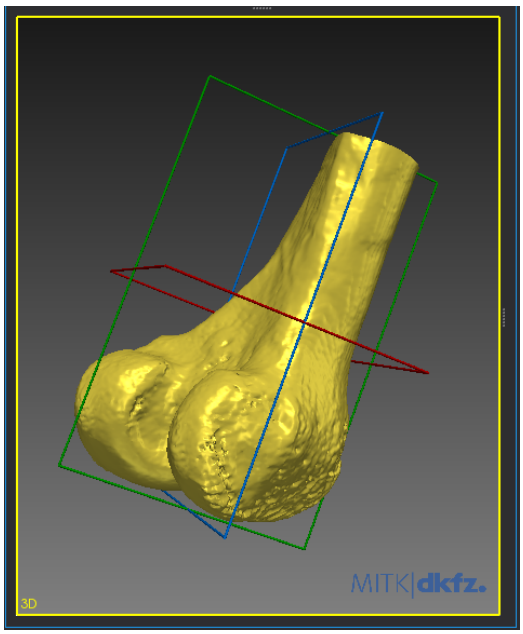
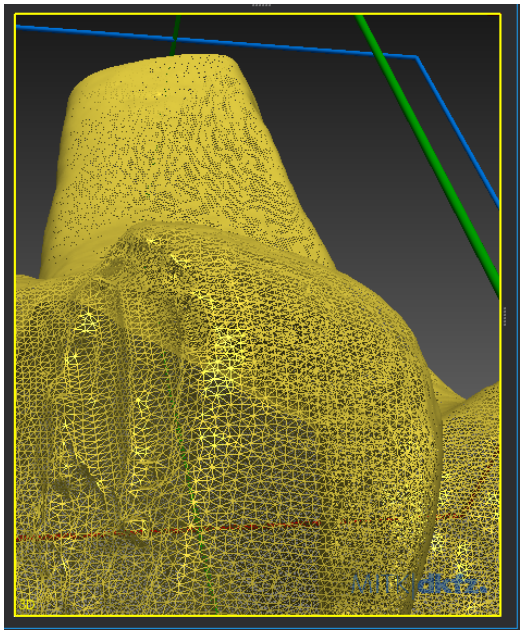


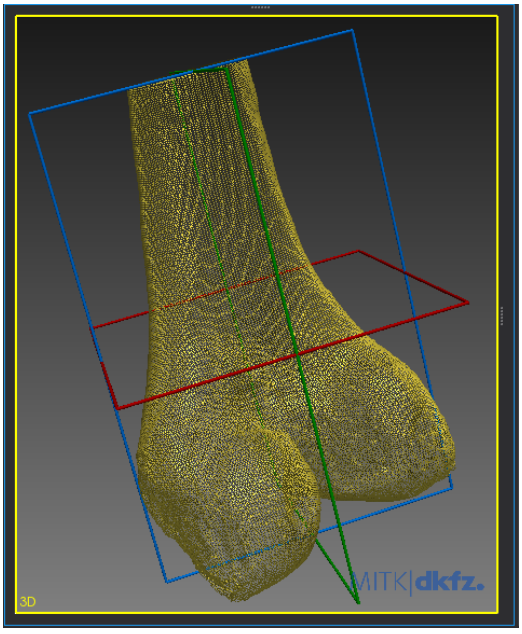
Figura 22: Ventana principal en modo Standard Display



(a) Representación de PolyData en estilo “Surface”.



(b) Representación de PolyData en estilo “Wireframe”.



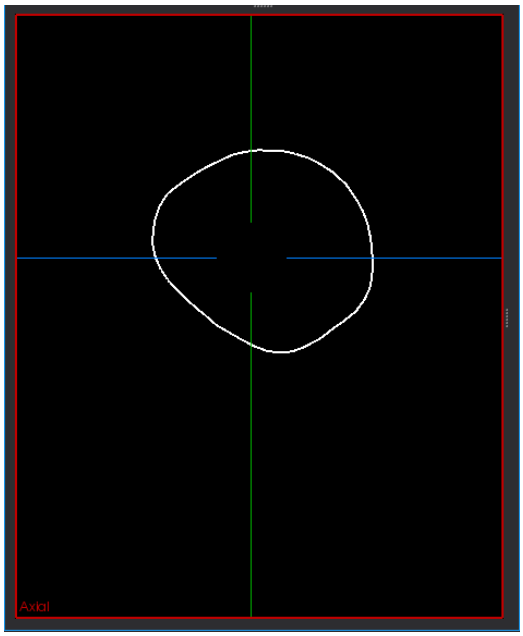
(c) Representación de PolyData en estilo “Point Cloud”.

Figura 23: Representación de objetos STL en la ventana de MITK.

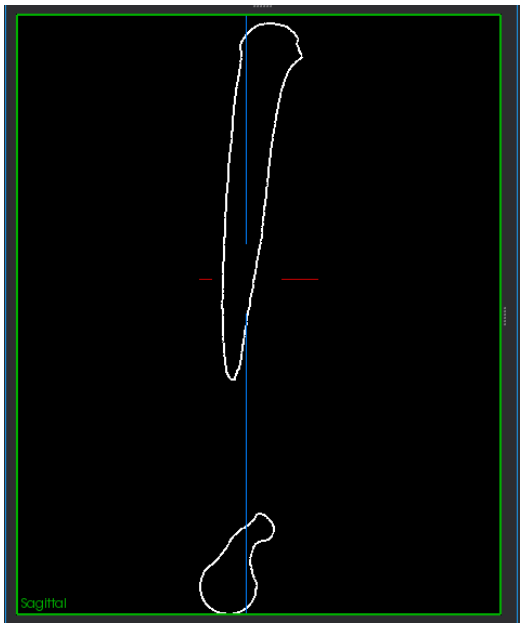
7.3.2. Ventanas Secundarias.

Además de la ventana principal, contamos con tres ventanas que reflejan vistas en 2-D, obtenidos a partir de los planos virtuales axial, sagital y coronal, correspondientes a los planos rojos, verde y azul en la ventana principal observable en la figura 22. Las ventanas secundarias están vinculadas con la principal en una relación bidireccional. Esto último significa que, desplazarse en cualquier ventana secundaria producirá un

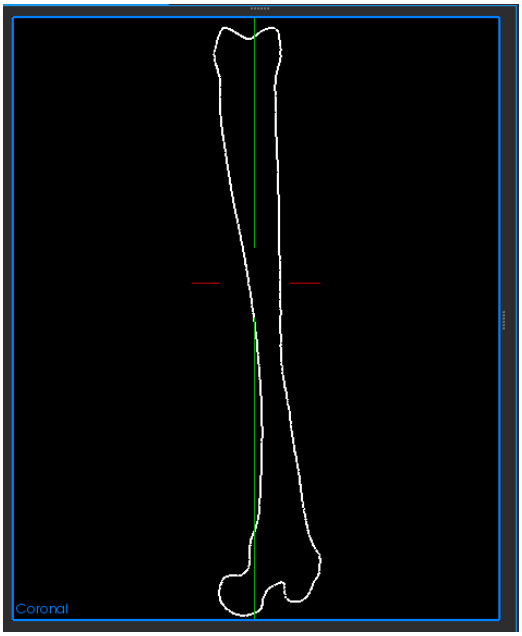
cambio en la posición del plano correspondiente en la principal y desplazar un plano en la ventana principal producirá un cambio en la ventana secundaria correspondiente. Estas 3 ventanas se pueden ver mejor en las figuras 24(a), 24(b), 24(c), página 29.



(a) Corte 2-D axial respecto a la ventana principal.



(b) Corte 2-D sagital respecto a la ventana principal.

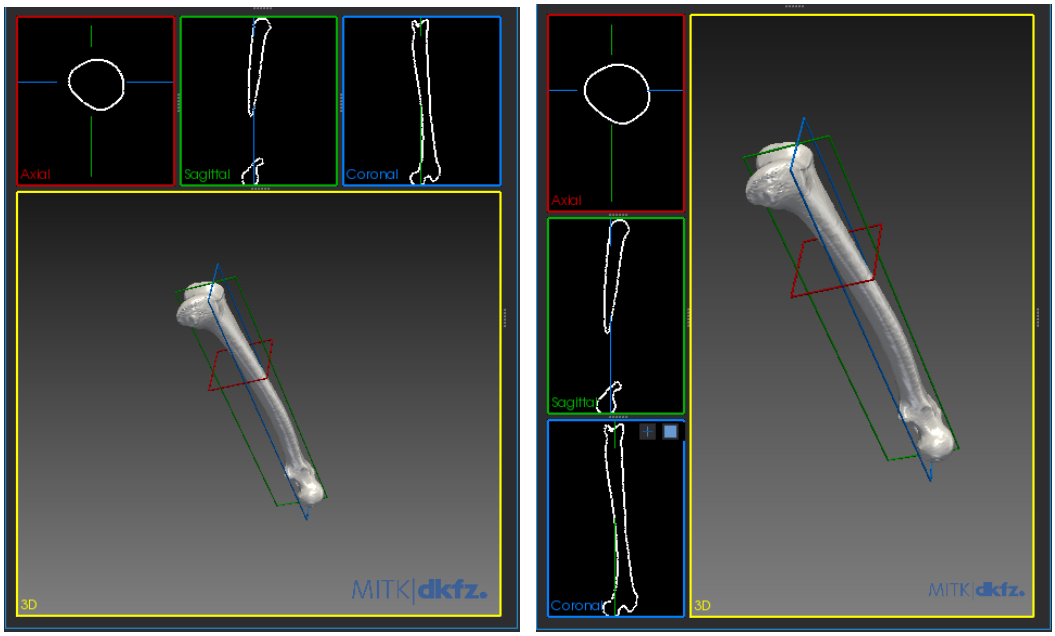


(c) Corte 2-D coronal respecto a la ventana principal.

Figura 24: Ventanas bidimensionales, vinculadas directamente con la ventana principal.

7.3.3. Layout de las ventanas

La figura número 22 ubicada en la página 27 muestra las 4 ventanas en lo que se denomina "Standard Display". La ventana principal se encuentra en la parte inferior derecha de la visualización y las ventanas 2-D se encuentran rodeándola. Podemos seleccionar otros tipos de organizaciones de ventana, por ejemplo mostrar únicamente la ventana principal o mostrar únicamente alguna de las secundarias. Otros Layouts incluyen mostrar la ventana principal debajo y las ventanas secundarias por arriba o la ventana principal a la derecha y las secundarias a un costado. Obsérvese estas variantes en las figuras 25(a) y 25(b) ubicadas en la página 30.



(a) Organización: ventana principal por debajo. (b) Organización: ventana principal a un costado.

Figura 25: Tipos de layout de ventanas.

Por último, es posible aplicarle un estilo configurable a la ventana principal como fue explicado anteriormente. Esto permitió tener un par de ejes y un sistema de coordenadas visible en la ventana. Obsérvese esto en la figura número 26 ubicada en la página 31.

7.4. Visualización múltiple

En caso de que se visualicen múltiples objetos en la ventana principal, es posible aplicar los conceptos y características mencionadas anteriormente a cada uno de los objetos por separado. Esto nos permite por ejemplo, representar Poly Datas (estructura de VTK para representar objetos) diferentes con varios colores para distinguirlos y combinar estilos de representación en una misma ventana. Por último, los cortes 2-D de cada objeto se distinguen a partir de su color. Obsérvese esto en las figuras número 27(a) y 27(b) ubicadas en la página 32.

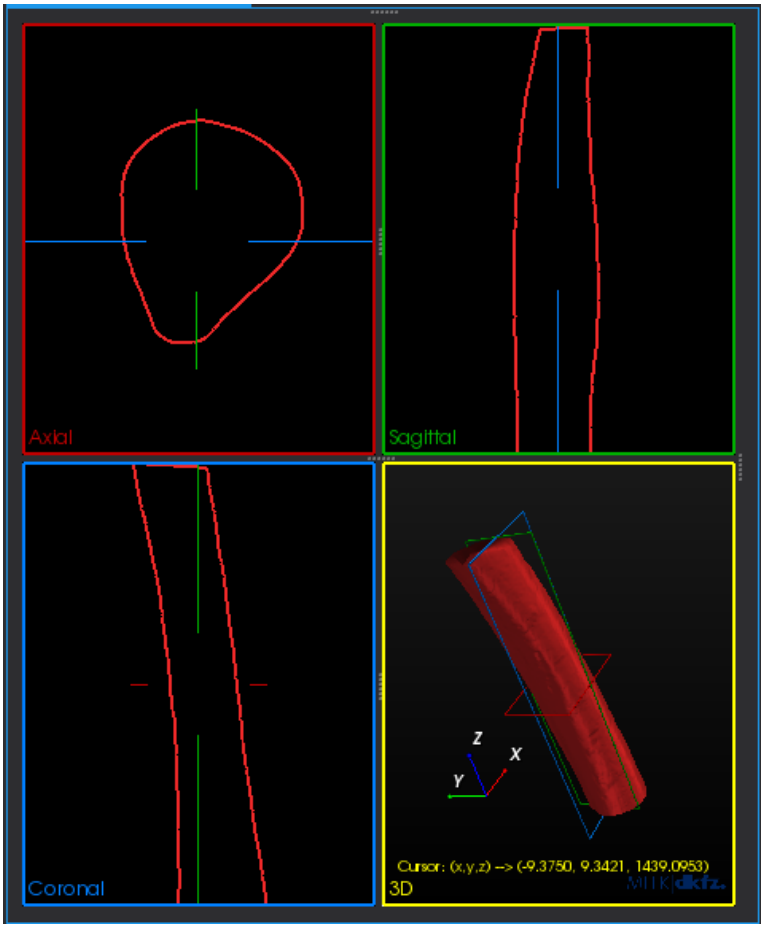
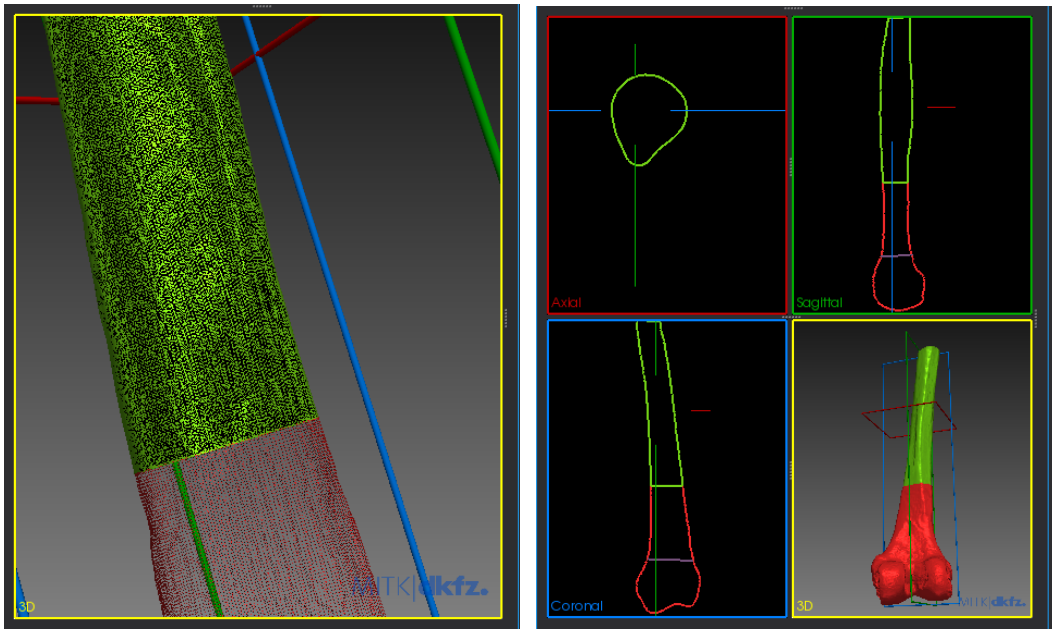


Figura 26: Layout con estilo configurable en la ventana principal.



(a) Distintos estilos de representación en simultáneo. (b) Los objetos en la ventana principal se observan en color en las ventanas secundarias.

Figura 27: Visualización con múltiple representación y estilos.

8. Desarrollo del plugin de registración.

El plugin desarrollado, que acopla nuevas Views a la WorkBench, se denomino **"register"**. Para crear el mismo, se generó una carpeta dentro del directorio del proyecto en la carpeta de plugins. Dicha carpeta posee el nombre del proyecto seguido del plugin: "template.register". Este plugin fue habilitado para ser utilizado según se especificó en la sección de "Configuración de plugin".

Para crear este plugin se requieren 4 archivos:

1. Archivo de configuración ".cmake".
2. Archivo de configuración "CMakeLists.txt".
3. Archivo activador de plugin.
4. Header activador de plugin.

En el primer archivo files.cmake vincularemos los archivos ".ccp", headers ".h" y archivos de GUI ".ui" que utilizamos durante el desarrollo del plugin:

```
set(CPP_FILES
  src/internal/template_register_PluginActivator.cpp
  src/internal/registerView.cpp
)

set(UI_FILES
  src/internal/registerViewControls.ui
)

set(MOC_H_FILES
  src/internal/template_register_PluginActivator.h
  src/internal/registerView.h
  src/internal/MultiThreading.h
)
```

En el archivo CMakeLists.txt crearemos el plugin, lo vincularemos con módulos (en este caso se utiliza un modulo de widgets de Qt y una librería de gráficos) y con su código fuente (en este caso ubicado en la carpeta "src"):

```
project(template_register)

mitk_create_plugin(
  EXPORT_DIRECTIVE REGISTRATION_EXPORTS
  EXPORTED_INCLUDE_SUFFIXES src
  MODULE_DEPENDS PRIVATE MitkQtWidgetsExt GraphicsLib
)
```

El archivo "template register PluginActivator.cpp", junto con su header, crean las views del plugin que aparecerá en la aplicación. Por tanto, vinculan el plugin creado con la Workbench basada en Blueberry y CTK que gestiona la creación del plugin. En el archivo .cpp definimos:

```
#include      template_register_PluginActivator     . h
#include    registerView   . h

void template_register_PluginActivator::start(ctkPluginContext* context)
{
  BERRY_REGISTER_EXTENSION_CLASS(registerView, context)
}

void template_register_PluginActivator::stop(ctkPluginContext*)
{
}
```

Los archivos "registerView.cpp" y "registerView.h" contienen el código del Plugin propiamente dicho y serán explicados en breve.

El archivo header "template register PluginActivator.h" definirá el activador del plugin y ciertas funciones de CTK necesarias para que éste funcione. El activador de este plugin en particular es definido a partir del activador base de CTK "ctkPluginActivator". Dentro de éste se define la identificación del plugin, en este caso "template register", junto con las funciones que crearán la View para este plugin. El objeto "ctkPluginContext" permite que nuestro plugin interactúe o sea afectado por otros métodos propios de CTK.

```
#ifndef template_register_PluginActivator_h
#define template_register_PluginActivator_h

#include <ctkPluginActivator.h>

class template_register_PluginActivator
  : public QObject,
    public ctkPluginActivator
{
  Q_OBJECT
  Q_PLUGIN_METADATA(IID "template_register")
  Q_INTERFACES(ctkPluginActivator)

public:
  void start(ctkPluginContext* context);
  void stop(ctkPluginContext* context);
};

#endif
```

8.1. Creación del plugin register

El desarrollo del plugin está volcado en los archivos:

- registerView.cpp

- registerView.h
- registerViewControls.ui

Los primeros dos archivos contienen el código fuente del plugin. El último archivo es la interfaz gráfica desarrollada en Qt. El resto de este informe y sus incisos están dedicados al desarrollo de este plugin.

8.2. Declaración de la clase registerView

En el inciso denominado "Estructura del proyecto" se elaboró sobre el concepto de View en la Workbench. En el header registerView.h se define la View destinada al Plugin en desarrollo. Como punto de partida se define la clase registerView:

```
class registerView : public QmitkAbstractView
{
Q_OBJECT
```

La View debe heredar de la clase "QmitkAbstractView", la cual es una clase base para manipular Views de aplicaciones de BlueBerry, recordando que las dos Views principales son las de la Interfaz de registración y el Data Manager. Es necesario que QOBJECT sea un atributo de la clase ya que necesitamos utilizar el método de signal-slot. Continuando:

```
public:

static const std::string VIEW_ID;

static const std::string SURFACE_NAME;

registerView();
~registerView() override;
```

En el scope público de la clase definimos el ID y nombre de la vista (puede haber más de una vista en una página de la Workbench, por lo que debe haber un número que identifique la View en cuestión unívocamente). Se define además un constructor y destructor de la clase de manera estándar. Continúo en el scope público definiendo la siguiente función:

```
void CreateQtPartControl(QWidget* parent) override;
```

La función anterior es la encargada de crear una View de Blueberry pero vinculada a Qt, es decir, una "berryQtViewPart". En términos simples, crea una vista que luego se podrá ver en la interfaz de la Workbench. En el scope protected de la clase se definen los slots de esta clase según lo explicado en la sección de signal-slot de este informe (inciso 5.4.1):

```
protected slots:

void OnSelectedSurfaceChanged(const mitk::DataNode *node);
void main_algorithm();
void confirm_selected();
void configure_window_interactor_renderer();
void DisplayOptimal(Ui::registerViewControls controls,double R,double T,
                    unsigned int iter,double grad_tol, double value_tol,double epsilon);
```

Los slots anteriores corresponden a funciones que utiliza la vista y serán explicadas en la sección 8, sin embargo es conveniente mostrar su existencia aquí. Continúo con el scope private de la clase:

```
private:

void SetFocus() override;

void OnSelectionChanged(berry::IWorkbenchPart::Pointer source,const QList<mitk::DataNode::
    Pointer>& dataNodes) override;
```

El primer método, denominado “SetFocus”, sirve para activar una parte de la vista en la workbench. La workbench invoca este método de manera automática sin intervención del usuario, pero es necesario definirla explícitamente. Luego la función “OnSelectionChanged” es invocada cada vez que cambia la selección de objetos en el Data Manager. Esto será explicado en la sección 8. Por último, volviendo al scope público se le agrega la interfaz gráfica de Qt como atributo a la clase registerView. Esto es conveniente cuando en la sección 8.5 se defina la implementación del método multithreading, el cual se encuentra en este mismo archivo header.

```
public:
Ui::registerViewControls mControls;
};
```

8.3. Configuración del escenario virtual

La primera configuración fue sobre el escenario virtual, es decir, el aspecto del ambiente de visualización, interactor y renderer. La descripción se encuentra en los snippets de código siguientes.

Primero obtenemos el interactor correspondiente a la ventana principal, la cual fue creada con el nombre “3d”:

```
QVTKInteractor* threeD_interactor = GetRenderWindowPart()->GetQmitkRenderWindow("3d")->
    GetInteractor();
```

Luego, deseamos que la ventana de visualización tenga un par de ejes cartesianos que guíen las direcciones de movimiento. Para esto, definimos un actor de tipo Axes y configuramos la longitud de cada eje, el tipo de terminación y línea:

```
vtkSmartPointer<vtkAxesActor> axis = vtkSmartPointer<vtkAxesActor>::New();
axis->SetNormalizedTipLength(0.1,0.1,0.1);
axis->SetTipTypeToSphere();
axis->SetShaftTypeToLine();
axis->SetNormalizedShaftLength(0.99,0.99,0.99);
```

Esta representación de los ejes cartesianos se debe colocar en una widget, para así poder manipular el interactor de la ventana “3d”.

```
vtkOrientationMarkerWidget *widget_orientation = vtkOrientationMarkerWidget::New();
widget_orientation->SetOutlineColor( 0, 0, 0 );
widget_orientation->SetOrientationMarker(axis);
widget_orientation->SetInteractor(threeD_interactor);
widget_orientation->SetViewport( 0.0, 0.0, 0.5, 0.5);
widget_orientation->SetEnabled( 1 );
widget_orientation->InteractiveOn();
vtkNamedColors *colores = vtkNamedColors::New();
```

Luego, nos interesa mostrar las coordenadas que tocamos en la pantalla. Para esto definimos un actor de tipo “textActor” que alberga una lista de strings.

```
vtkSmartPointer<vtkTextActor> textActor = vtkSmartPointer<vtkTextActor>::New();
textActor->GetTextProperty()->SetFontSize ( 9 ); textActor->SetPosition ( 10, 20 );
textActor->SetInput ( "Cursor: inactivo" );
textActor->GetTextProperty()->SetColor(colores->GetColor3d("Yellow").GetData());

GetRenderWindowPart()->GetQmitkRenderWindow("3d")->GetVtkRenderWindow()->SetCurrentCursor(
    VTK_CURSOR_CROSSHAIR);
```

Obtenemos el renderer propio de la ventana “3d” para renderizar los actores agregados.

```
mitk::BaseRenderer::Pointer threeD_renderer = GetRenderWindowPart()->GetQmitkRenderWindow("3d
    ")->GetRenderer();
threeD_renderer->GetVtkRenderer()->AddActor2D(textActor);
threeD_renderer->GetVtkRenderer()->SetBackground(colores->GetColor3d("black").GetData());
```

Por último, para actualizar la posición seleccionada en la visualización, debemos crear un Point Picker que nos permita muestrear coordenadas globales. El Picker se le asigna al interactor. A este último le asignamos un estilo de interacción particular dado por "MouseInteractorStyle", el cual será detallado en la sección 8.4.

```
vtkSmartPointer<vtkWorldPointPicker> worldPointPicker = vtkSmartPointer<vtkWorldPointPicker>::
    New();
threeD_interactor->SetPicker(worldPointPicker);
vtkSmartPointer<MouseInteractorStyle> estilo = vtkSmartPointer<MouseInteractorStyle>::New();
estilo->PositionActor = textActor;
threeD_interactor->SetInteractorStyle(estilo);
}
```

La figura 28 muestra la ventana de visualización configurada.

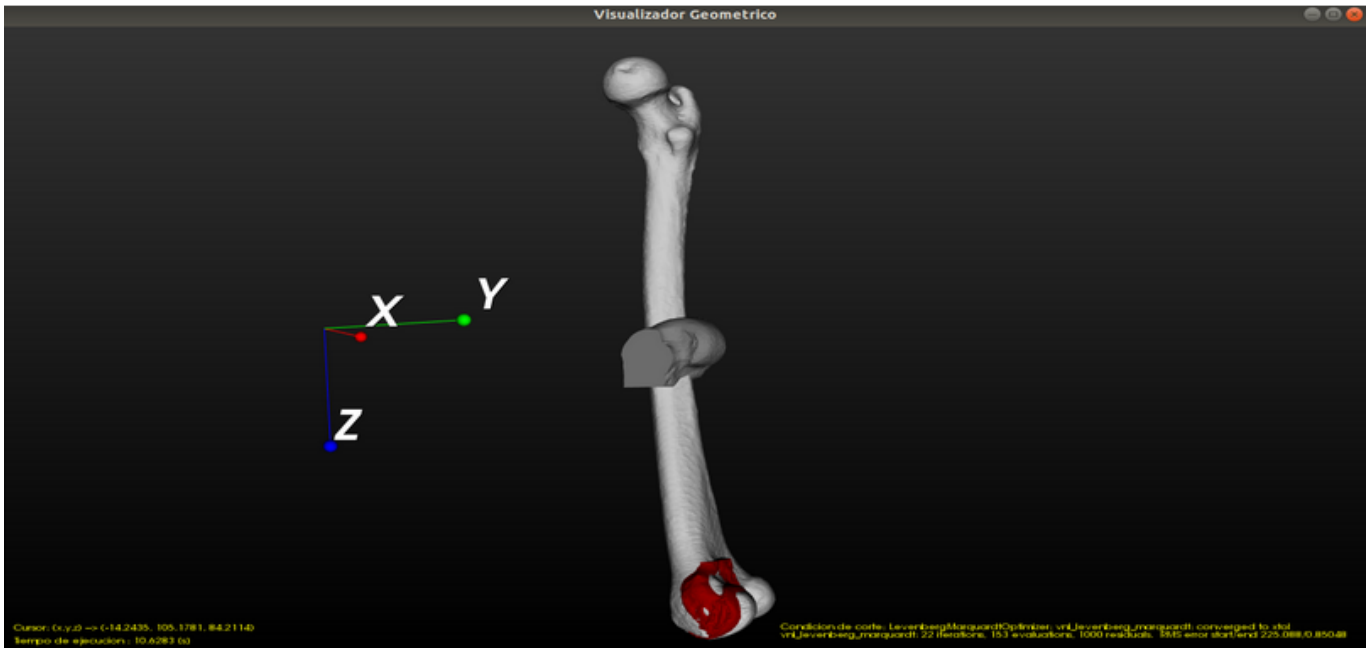


Figura 28: Ventana de visualización principal del proyecto.

8.4. Implementación de un estilo de interacción propio

Por motivos relacionados con la registración de objetos en la escena, fue necesario tener una configuración propia del estilo de interacción. Para esto, se propuso generar una subclase de un estilo de interacción ya existente en VTK llamado "Trackball Camera Interactor Style". Éste permite que, a través de la manipulación de la cámara (rotación, zoom, etc), el usuario pueda cambiar el punto de vista de la escena. En este tipo de interacción hay una proporción entre el movimiento del mouse y el movimiento de la cámara. Con la configuración elegida podremos controlar la escena con los comandos:

1. Rotación de la cámara: pequeños movimientos del botón izquierdo del mouse generan cambios pequeños en la rotación de la cámara alrededor de su foco.
2. Zoom de la escena: el click derecho aplica zoom en la escena.
3. Traslación de actores: se puede desplazar un objeto en escena a otra coordenada presionando la tecla shift junto con el click izquierdo.

Los siguientes extractos de código muestran cómo se configuró el nuevo interactor a partir del estilo Trackball Camera.

Creamos la clase MouseInteractorStyle, la cual hereda de vtkInteractorStyleTrackballCamera. La línea "vtkTypeMacro" nos permite vincular la superclase con su subclase.

```
class MouseInteractorStyle : public vtkInteractorStyleTrackballCamera
{
public:
static MouseInteractorStyle* New();
vtkTypeMacro(MouseInteractorStyle,vtkInteractorStyleTrackballCamera)
```

Luego vamos a modificar la respuesta al evento "click izquierdo presionado" redefiniendo la función OnLeftButtonDown() y agregándole internamente un Point Picker que genere un muestreo de la posición del evento (en coordenadas globales) y la guarde en un arreglo.

```
virtual void OnLeftButtonDown()
{
this->Interactor->GetPicker()->Pick(this->Interactor->GetEventPosition()[0],
this->Interactor->GetEventPosition()[1]
this->Interactor->GetRenderWindow()->GetRenderers()->GetFirstRenderer());
double picked[3];
this->Interactor->GetPicker()->GetPickPosition(picked);
```

Luego esa variable con las coordenadas se asignará a un vtkTextActor (atributo de la nueva clase creada), el cual es posible renderizar. Por último, cada vez que el estilo de interacción sea utilizado, se actualizará dicho actor con las coordenadas obtenidas por el picker.

```
std::ostringstream text;
text << "Cursor: (x,y,z) --> ("
<< std::fixed << std::setprecision(4)
<< picked[0] << ", " << picked[1] << ", " << picked[2]<<")";
this->PositionActor->SetInput(text.str().c_str());
vtkInteractorStyleTrackballCamera::OnLeftButtonDown();
}
vtkTextActor *PositionActor;
};
vtkStandardNewMacro(MouseInteractorStyle)
```

Para ser más precisos sobre la implementación anterior, cabe destacar y repasar ciertas cuestiones: la función "OnLeftButtonDown()" de la clase original (genera un proceso cuando se detecta un click con el mouse izquierdo) fue redefinida para obtener las coordenadas globales de un punto de algún objeto en la ventana de visualización. Esto se hace mediante el uso de un objeto de vtk denominado "Point Picker" (propiedad del renderer seleccionado en la ventana). Este picker nos permite obtener las coordenadas de un actor en escena mediante ray casting, es decir, el picker disparará un rayo en la ventana en las coordenadas (x,y) donde se presione el boton izquierdo. Este rayo interseca al actor que se desea seleccionar y devuelve la coordenada z de la superficie del mismo. Cada objeto en la escena posee una propiedad denominada bounding box, que define el entorno del objeto para distinguirlo de otro. Dicho esto, el picker podría retornar más de una coordenada "z" ya que el rayo pudo haber intersecado dos objetos diferentes. El picker devuelve una lista sin ordenar de las coordenadas de los objetos intersecados y devuelve la del objeto más cercano a la cámara según el vector que define el rayo. Luego se anuncian las coordenadas obtenidas mediante línea de comando y para la escena virtual se inserta un actor de tipo "Text Actor" conteniendo las coordenadas.

8.5. Implementación del Multithreading

Como se ha mencionado anteriormente, el objetivo principal de esta implementación es separar el GUI Loop de las instancias de registración.

Para la implementación se decidió inicialmente seguir un esquema de Worker-Thread junto con el esquema de Signal-Slot. Cada programa tiene un thread cuando es ejecutado, a éste lo llamamos en Qt, "Main thread". La interfaz de Qt corre en este thread. Todas las widgets asociadas a la GUI corren en este hilo principal. A un thread secundario se lo denomina "Worker Thread" porque se usa para quitarle carga de procesos al thread principal.

En el siguiente script, se demuestra cómo fue implementada inicialmente esta lógica utilizando la clase Worker:

```
class Worker : public registerView
{
  Q_OBJECT
private:
    Ui::registerViewControls Controls;

public:
    Worker(Ui::registerViewControls controls){
    Controls = controls;
    }

public slots:
    void threaded_registration();
};
```

En síntesis, se crea una clase. La misma debe contener características de QObject debido a que esto permite la interacción Signal-Slot. Además, recibe una instancia de la GUI. Por último, se define un Slot correspondiente a la función que va a realizar la registración. El método threaded registration se implementó según:

```
void Worker::threaded_registration(){

double R_scale = this->Controls.r_scale_input->value();
double T_scale = this->Controls.t_scale_input->value();
unsigned int max_iter = (unsigned int)this->Controls.max_iter_input->value();
double grad_tol = this->Controls.grad_tol_input->value();
double value_tol = this->Controls.value_tol_input->value();
double epsilon = this->Controls.epsilon_input->value();

DisplayOptimal(this->Controls,R_scale,T_scale,max_iter,grad_tol,value_tol,epsilon);
}
```

En resumen, la función anterior es propia de cada instancia de la clase Worker y permite pasarle los parámetros de configuración de la registración que se ingresan mediante la GUI. Por último, el método llama a "Display Optimal" para iniciar la registración.

En el siguiente script se muestra cómo se puede implementar un esquema de Worker-Thread basado en la clase anterior:

```
Worker *worker_1 = new Worker(mControls);

QThread *thread_1 = new QThread;
thread_1->setObjectName("Thread number 1");

connect(thread_1,&QThread::started, worker_1, &Worker::threaded_registration);
connect(thread_1,&QThread::finished,worker_1,&QObject::deleteLater);

worker_1->moveToThread(thread_1);
thread_1->start();
```

En síntesis, el script anterior crea una instancia de Worker y le asigna mControls (una instancia de la GUI). Luego se crea un Thread. Mediante el esquema Signal-Slot se conecta el thread con el worker. Cuando el thread es iniciado se dispara la señal "started" y el worker comienza la registración mediante el método "threaded registration". Luego se emite una señal cuando la ejecución en el thread termina, de manera que el worker asignado a éste se elimine. Por último, mediante la función "movetoThread" se puede asignar el worker instanciado al thread creado. Con la función "start" se inicia la ejecución en el thread.

8.5.1. Implementación mediante Thread Pool

En estadíos más avanzados del proyecto surgió la necesidad de administrar los threads, de manera que el mismo trabajo no sea procesado por dos threads separados. También se deseó que, cuando un thread se encuentre libre (por haber terminado de procesar un trabajo por ejemplo) se le asigne una tarea de manera

automática. Para esto, se utilizó la clase de Qt llamada "QtConcurrent". Ésta, nos permite administrar una "Pool" de threads ocupándose de los problemas mencionados anteriormente. Cada aplicación de Qt (en este caso TempleApp) posee una pool de threads propia, siendo uno de los threads de la pool el que mantiene la ejecución del GUI Loop. La función "QtConcurrent:run" permite asignar una función y los argumentos de esa función a un thread libre en una Pool determinada. Se puede crear una Pool propia aunque, si no se especifica nada, se utiliza la Pool default de la aplicación. Esto se muestra en la siguiente sección de código (parte de la función "main algorithm"):

```
mitk::DataStorage::SetOfObjects::ConstPointer rs = mControls.template_box_select->GetNodes();
for (unsigned int i = 0; i < rs->size(); i++){
mitk::DataNode::Pointer template_node=rs->GetElement(i);
if (template_node->GetName() == target_node->GetName() ){
std::cout<<"Not registering object with itself"<<std::endl;
}else{
//do stuff with DataNodes
QtConcurrent::run(this,&registerView::DisplayOptimal,template_node);
}
}
```

De la manera anterior se puede iterar sobre los nodos del Data Storage y asignar el nodo y la función de registración a la Pool de threads default de la aplicación Template App. En este caso la función DisplayOptimal pertenece a una clase (member function), por esto, la función run de QtConcurrent debe recibir una instancia de la clase relacionada con la función.

8.6. Inicialización de los nodos

Cuando se importan las superficies a utilizar en la aplicación, las mismas ingresan como un nodo en el Data Storage. Luego, cuando estos objetos se visualizan en las ventanas de la aplicación, aparecen en las coordenadas especificadas en su estructura de fondo (Surface). Esto quiere decir que las estructuras que deseamos alinear podrían aparecer muy lejos entre sí.

Con el objetivo de acercar estas estructuras, se desarrolló una función de inicialización que lleve a los objetos a una situación inicial más favorable. Para esto, se buscó centrar los objetos en el origen de coordenadas mediante una substracción de las coordenadas del centro de masa de cada objeto con sus coordenadas de aparición inicial. De esta manera, todos los objetos tendrán su centro de masa coincidente con el origen de coordenada de la ventana de visualización. El resultado de inicializar nodos se puede observar en la figura 29. Observar la implementación en los siguientes extractos:

Como primer punto, se selecciona el nodo activo y se crea el objeto de la transformación, el filtro de transformación y el centro de masa:

```
mitk::DataNode::Pointer node = this->mControls.initializing_box->GetSelectedNode();

vtkSmartPointer<vtkCenterOfmass> Origin = vtkSmartPointer<vtkCenterOfmass>::New();
vtkSmartPointer<vtkTransform> Transformacion_Centro = vtkSmartPointer<vtkTransform>::New();
vtkSmartPointer<vtkTransformPolyDataFilter> Transformacion_Centro_Filtro =
    vtkTransformPolyDataFilter::New();
```

Luego se extrae la superficie del nodo y el PolyData de la superficie. Se genera la superficie output:

```
mitk::Surface::ConstPointer surface_pointer = static_cast<mitk::Surface*>
        (node->GetData());

vtkSmartPointer<vtkPolyData> polydata = surface_pointer->GetVtkPolyData();

mitk::Surface::Pointer surface = mitk::Surface::New();
```

Luego se obtiene el centro de masa del PolyData mediante la clase "center of mass":

```
Origin->SetInputData(polydata);
Origin->SetUseScalarsAsWeights(0);
Origin->Update();
```

Se configura la transformación definiéndola como una operación que le restará, a cada coordenada del PolyData, las coordenadas del centro de masa del mismo. Se mapea la transformación al filtro de transformación y se actualiza:

```
Transformacion_Centro->Translate(-(Origin->GetCenter())[0],-(Origin->GetCenter())[1],-(Origin
    ->GetCenter())[2]);
Transformacion_Centro_Filtro->SetInputData(polydata);
Transformacion_Centro_Filtro->SetTransform(Transformacion_Centro);
Transformacion_Centro_Filtro->Update();
```

A partir del filtro, se obtiene el PolyData y la superficie centrada.

```
vtkSmartPointer<vtkPolyData> polydata_centrado=Transformacion_Centro_Filtro->GetOutput();
surface->SetVtkPolyData(polydata_centrado);
```

Por último, esta superficie se agrega a un nodo que irá al Data Storage. Previamente se configurará el nombre del nodo (el mismo que el nodo no decimado), se lo hará visible y se removerá el nodo anterior.

```
mitk::DataNode::Pointer centered_node = mitk::DataNode::New();
centered_node->SetData(surface);
centered_node->SetName(node->GetName());
centered_node->SetVisibility(true);
this->GetDataStorage()->Remove(node);
this->GetDataStorage()->Add(centered_node);
```

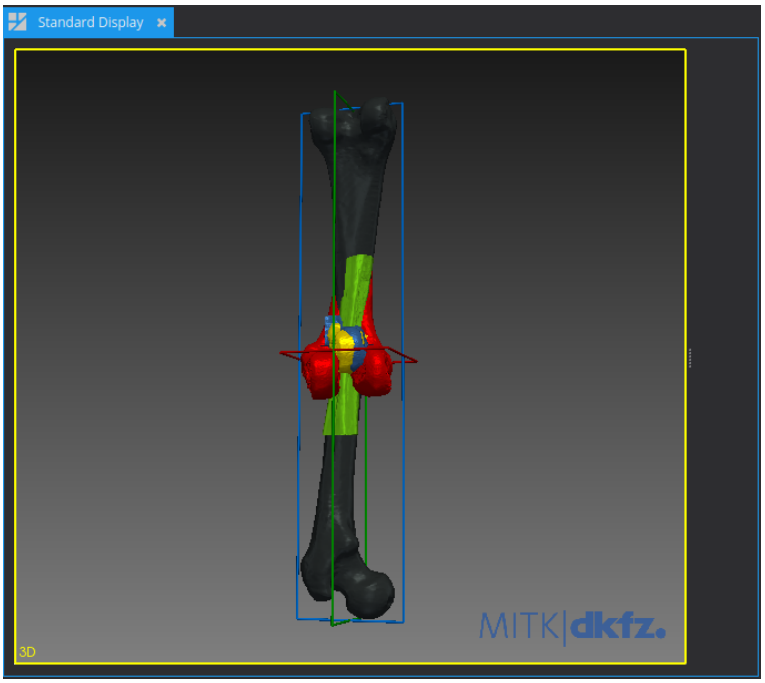


Figura 29: Ejemplo de la inicialización de múltiples nodos.

8.7. Reducción de densidad de los nodos

Un problema que se encontró temprano en el desarrollo de este proyecto fue el de velocidad de convergencia. Las estructuras STL (formato de las superficies utilizadas por este proyecto) se componen de vértices formando un mallado triangular o cuadrangular. En muchos casos, la cantidad de puntos es muy grande, en ocasiones superior 10.000. La complejidad del algoritmo ICP escala según el producto de la cantidad de puntos en ambas

nubes, es decir O(N1*N2). Al tratar de registrarse una cantidad muy grande de puntos, la complejidad escala muy rápido al pasar de estructuras pequeñas a grandes.

Debido a lo mencionado anteriormente, se buscó una forma de reducir la cantidad de vértices de las superficies a registrar mediante algún método de decimación. Para esto se utilizó un algoritmo validado que, configurado de manera correcta, podría utilizarse de manera efectiva. El objetivo es transformar la nube de puntos inicial en una de menor tamaño, pero que mantenga la forma inicial en secciones importantes como los bordes o concavidades.

Los parámetros a configurar para esta decimación son los siguientes:

- **Superficie Input**: estructura de nodos en mallado triangular o cuadrangular.
- **Time Step**: si deseamos tratar con una estructura 4-D (donde el tiempo avanza y la superficie cambia con el tiempo).
- **Número de vértices**: número de puntos en la superficie final con respecto a la inicial.
- **Gradation**: refleja la influencia de la curvatura en la decimación en una escala de [0,1], donde una escala alta significa que se van a evitar remover nodos donde se detecte curvatura o concavidad.
- **Subsampling factor**: es el factor o porcentaje de decimación.
- **Edge split**: este factor fue seteado a cero para no eliminar bordes, es decir, no discontinuarlos.
- **Optimization level**: este factor varía en el intervalo [0,1] y permite mantener la distancia entre la superficie original y la decimada de manera que, si el factor es alto, la distancia entre ambas se minimiza.
- **Boundary fixing**: permite realizar un padding con vértices extra para mantener los límites de las superficies originales.

Estos parámetros se ingresan en el decimador, junto con algunos cálculos previos que permiten estimar los vértices de la superficie input y output mediante el factor de decimación:

```
int max_vertex_nr = static_cast<int>(surface->GetVtkPolyData()->GetNumberOfPoints());
int vertex_nr = std::max(100, static_cast<int>(max_vertex_nr * (halving_factor * 0.01)));
int downsample_factor = 10;

mitk::ACVD::RemeshFilter::Pointer decimator = mitk::ACVD::RemeshFilter::New();

decimator->SetInput(surface);
decimator->SetTimeStep(0);
decimator->SetNumVertices(vertex_nr);
decimator->SetGradation(1);
decimator->SetSubsampling(downsample_factor);
decimator->SetEdgeSplitting(0.0);
decimator->SetOptimizationLevel(1.0);
decimator->SetForceManifold(false);
decimator->SetBoundaryFixing(true);
```

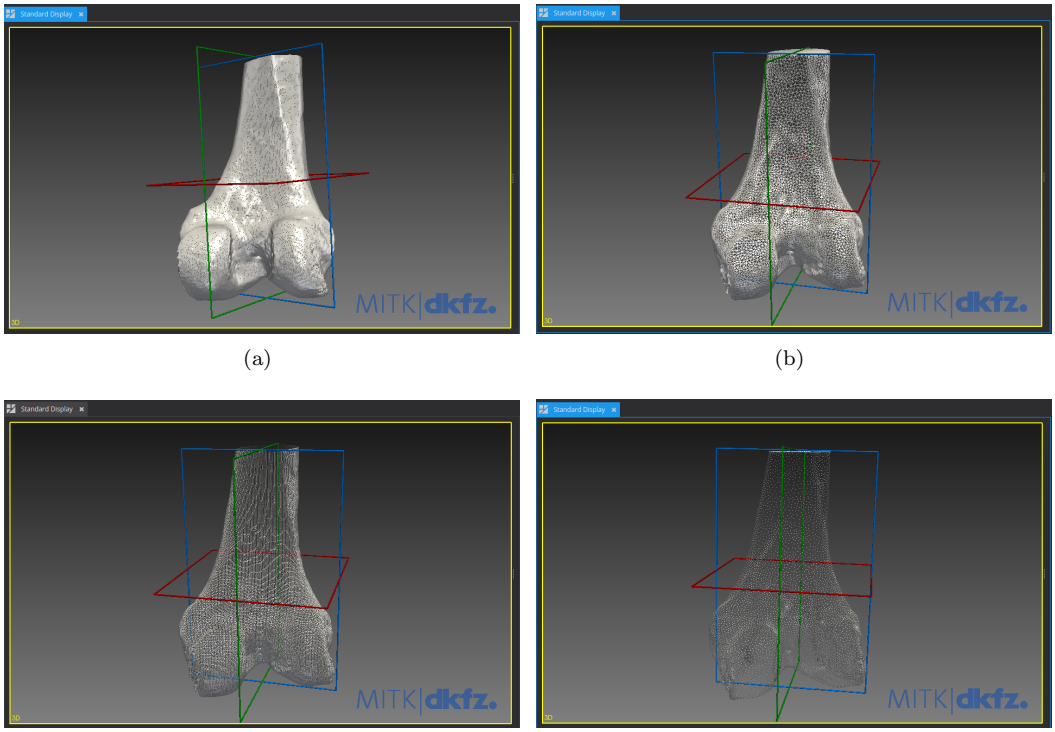
Mediante un esquema try and catch se actualiza el decimador para ejecutar el proceso:

```
try
{
  decimator->Update();
}
catch(const mitk::Exception& exception)
{
  MITK_ERROR << exception.GetDescription();
}
```

Por último, se obtiene y devuelve el output, es decir, el PolyData decimado:

```
mitk::Surface::Pointer decimated_surface = decimator->GetOutput();
vtkSmartPointer<vtkPolyData> decimated_polydata = decimated_surface->GetVtkPolyData();

return decimated_polydata;
```

Observar el resultado de la decimación de un nodo, en distintos estilos de visualización, en la figura 30.



(a) (b)

(c) (d)

Figura 30: Decimación de un nodo y visualización en distintos estilos.

8.8. Corrección manual

El método de registración propuesto tiene limitaciones. Es difícil rotar el objeto a registrar en ángulos grandes dado que, avanzar con un paso fijo pequeño nos lleva a caer en errores locales. Por esto, luego de inicializar los nodos es necesario realizar un ajuste manual en la inclinación del nodo a registrar con el nodo fijo. La alineación no debe ser exhaustiva sino que disminuya la necesidad de girar ángulos grandes (mayores a 30 grados). El ajuste manual permitirá desplazar los objetos y rotarlos respecto al origen de coordenadas o su propio centro de masa. Esto se implementó tomando los interactores base de cada objeto y ajustando el paso para cada transformación.

Las transformaciones posibles son las siguientes:

- Traslación lateral.
- Traslación hacia arriba y abajo.
- Traslación en profundidad.
- Rotación sobre eje X.
- Rotación sobre eje Y.
- Rotación sobre eje Z.

Obsérvese la implementación a continuación (solo se muestran las partes más significativas de cada método):

Para una traslación simple, se implementó la siguiente función (solo se muestra la traslación en un solo eje pues las demás son análogas). Se comienza con la obtención del nodo:

```
1 void registerView::control_up(){
2
```

```
QString node_tmp_1 = mControls.alignment_box->text();
QByteArray name_tmp_2 = node_tmp_1.toLocal8Bit();
const char *node_name = name_tmp_2.data();

mitk::DataNode::Pointer node = GetDataStorage()->GetNamedNode(node_name);
```

El nodo es transformado a Poly Data:

```
mitk::Surface::ConstPointer surface = static_cast<mitk::Surface*>(node->GetData());

vtkSmartPointer<vtkPolyData> polydata = surface->GetVtkPolyData();
```

Se configura una transformación de tipo traslación cuya escala y valor están definidos por la interfaz que manipula el usuario:

```
vtkSmartPointer<vtkTransform> up_transform = vtkSmartPointer<vtkTransform>::New();

vtkSmartPointer<vtkTransformPolyDataFilter> transform_filter =
vtkSmartPointer<vtkTransformPolyDataFilter>::New();

up_transform->Translate(-mControls.xyz_tscale->value(), 0, 0);

transform_filter->SetTransform(up_transform);
transform_filter->SetInputData(polydata);
transform_filter->Update();
vtkSmartPointer<vtkPolyData> up_polydata=transform_filter->GetOutput();
```

Se mappea la superficie de salida sobre un nodo nuevo, y se definen ciertas propiedades para este nodo:

```
mitk::Surface::Pointer up_surf = mitk::Surface::New();
up_surf->SetVtkPolyData(up_polydata);

mitk::DataNode::Pointer up_node = mitk::DataNode::New();
up_node->SetData(up_surf);
up_node->SetName(node->GetName());
up_node->SetColor(1,0,0);
up_node->SetOpacity(0.6);
up_node->SetSelected(true);
up_node->SetVisibility(true);
```

Se elimina del Data Storage el nodo original y se agrega el actualizado, terminando el proceso:

```
string clean_id = node->GetName() + "-eliminate";
node->SetName(clean_id);
node->Update();

this->GetDataStorage()->Remove(node);
this->GetDataStorage()->Add(up_node);
}
```

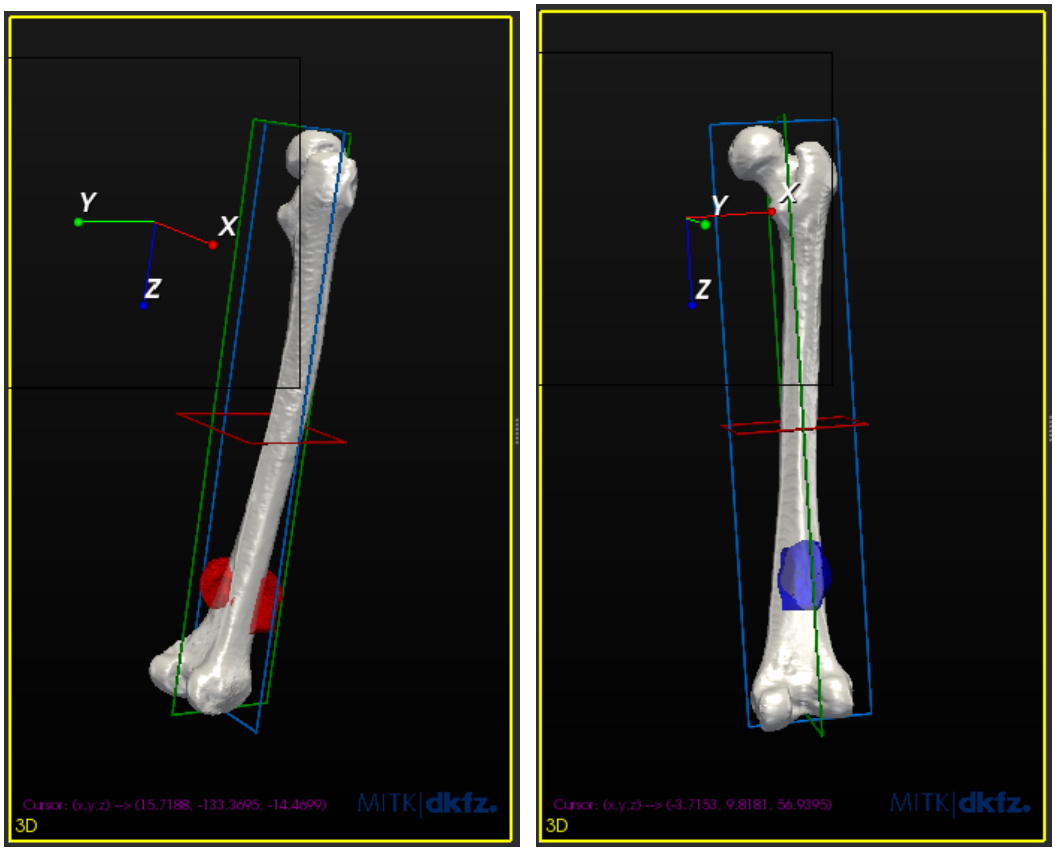
Para implementar una rotación se debe proceder de manera similar, pero definiendo un centro de rotación (el del objeto) y anclando la rotación en las coordenadas de dicho centro:

```
rotated_node->SetFloatProperty("AffineBaseDataInteractor3D.Anchor Point X",
node->GetData()->GetGeometry()->GetCenter()[0]);

rotated_node->SetFloatProperty("AffineBaseDataInteractor3D.Anchor Point Y", node->GetData()->
    GetGeometry()->GetCenter()[1]);

rotated_node->SetFloatProperty("AffineBaseDataInteractor3D.Anchor Point Z", node->GetData()->
    GetGeometry()->GetCenter()[2]);
```

Por último, para detener el proceso de alineamiento y restablecer las propiedades originales del nodo:

```
void registerView::stop_prealignment(){

QString node_tmp_1 = mControls.alignment_box->text();
QByteArray name_tmp_2 = node_tmp_1.toLocal8Bit();

const char *node_name = name_tmp_2.data();

mitk::DataNode::Pointer node = GetDataStorage()->GetNamedNode(node_name);

node->SetColor(1,1,1);
node->SetOpacity(1);
node->Update();
}
```

Obsérvese en la figura 31, un ejemplo de corrección manual de nodos.



(a) (b)

Figura 31: El color rojo representa la traslación del objeto, mientras que el azul la rotación.

8.9. Algoritmo de registración utilizado

El algoritmo de registración utilizado se basa en nube de puntos. Para esto se utilizo la clase "itkPointSetToPointsetRegistration". Esta clase permite definir una clase base de registración por puntos, a la cual se le pueden configurar los siguientes parámetros:

1. **Transformación**: la operación matemática que alineará una nube de puntos con la otra.
2. **Métrica**: ésta compara ambos set de puntos. El objetivo del método de registración es encontrar los parámetros de transformación que optimicen esta métrica.

3. **Optimizador**: éste guía el método de registración hasta la métrica óptima.

Esta clase utiliza el sistema de coordenadas de una nube de puntos fija como referencia. Luego, busca una transformación que genere un mapeo hacia el sistema de coordenadas de una nube de puntos móvil. Para lograr esto, se calcula una métrica de manera continua para comparar ambas nubes de puntos. Para que esta clase pueda ser instanciada, debemos definir explícitamente una transformación, una métrica y un optimizador, los cuales serán mencionados en los siguientes incisos de esta misma sección.

8.9.1. Transformación

La transformación seleccionada es la de ángulos de Euler. ITK nos brinda la clase "Euler 3D Transform". Esta transformación aplica una rotación y traslación a una nube de puntos dados 3 ángulos de Euler y un vector de traslación en 3 dimensiones. Para definir la rotación, se deben especificar las coordenadas de un centro de rotación.

Los parámetros de esta transformación pueden configurarse utilizando la función SetParameters(), mediante un arreglo de 6 elementos, donde T_i indican traslaciones en los 3 ejes cartesianos y $R_{\theta ic}$ indica cada rotación en ángulo de Euler con centro de rotación en "c":

$$F(R_{\theta xc}\hat{i}, R_{\theta yc}\hat{j}, R_{\theta zc}\hat{k}, T_x\hat{i}, T_y\hat{j}, T_z\hat{k}) = F(\vec{R}_{\theta\vec{x}c}, \vec{T}_{\vec{x}}) = F(\vec{P})$$

Esto se puede resumir en una matriz denominada Matriz de Transformación de Euler, en donde se encuentran como elementos estos 6 parámetros. Esta matriz puede insertarse como un parámetro a la transformación de ITK:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Dicha matriz representa la traslación, escalamiento y rotación. La submatriz de 3x3 formada por los elementos de a_{11} a a_{33} contiene información sobre los ángulos de Euler y la escala, mientras que la última columna contiene el vector de traslación.

La figura 32 muestra una analogía entre los ángulos de Euler y el movimiento de un avión.

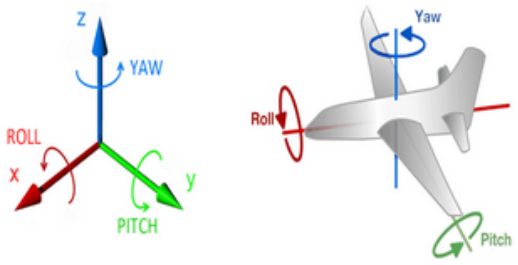


Figura 32: Simple demostración de los ángulos de Euler. Fuente: Chrobotics.

8.9.2. Métrica

Se decidió adoptar la distancia euclídea como métrica para comparar las nubes de punto N_1 y N_2 con cantidad de puntos n_1 y n_2 respectivamente, siendo $N_1 \leq N_2$ y x_{2c} el punto más cercano de la nube N_2 asociado al punto x_{1i} de la nube N_1:

$$E(\vec{N}_1, \vec{N}_2) = \sqrt{\sum_{i=1}^{n_1} (x_{1i} - x_{2c})^2}$$

La clase de "itkEuclideanDistancePointMetric" puede asociarse directamente con el registrador de ITK seleccionado. Ésta computa la distancia mínima entre la nube de puntos fija y móvil. No es necesario establecer una correspondencia entre ambas nubes de puntos. Esto último se debe a que, como estimación, se crea un vector que contiene el punto más cercano para cada punto de la nube opuesta. Para lograr mayor velocidad en el cálculo, ITK recomienda utilizar el set de puntos móvil como aquel con la menor cantidad de puntos, lo cual resulta intuitivo.

8.9.3. Optimizador

El optimizador es el encargado de alterar los parámetros de la transformación para lograr una métrica óptima. Se debe lograr calcular los parámetros que verifican la siguiente ecuación:

$$E^2[\vec{N}2, F(\vec{R}_{\theta\vec{x}c}, \vec{T}_{\vec{x}})\{\vec{N}1\}] = 0$$

Es decir, la nube de puntos fija debe ser igual a la versión transformada de la nube móvil. El optimizador elegido deberá modificar los 6 parámetros hasta lograr dicha condición.

El algoritmo matemático seleccionado para este trabajo es el método de "Levenberg-Marquadt" (abreviado LMA en ingles), también conocido como "Método de los cuadrados mínimos amortiguados" (abreviado DLS en inglés), utilizado para resolver cuadrados mínimos para funciones no lineales. Una limitación de este método es que tiene dificultades para hallar mínimos globales de manera eficaz, si la inicialización de los parámetros no es buena. Por esto, es importante proveerle al algoritmo una posición inicial favorable.

Matemáticamente lo que deseamos hallar se puede expresar como:

$$argmin_{\vec{P}} \sum_{i=1}^{m} [N_{2i} - F(\vec{P})\{N_{1c}\}]^2$$

Esta minimización se resuelve mediante un proceso iterativo. Inicialmente, es necesaria una estimación del vector de parámetros . En cada iteración del algoritmo, el vector de parámetros se incrementa por un factor determinado por el optimizador de métrica. En la iteración "k" tendremos:

$$\vec{P}_k = \vec{P}_{k-1} + \vec{\delta}$$

El paso o "step" se determina a partir de la linealización de la función:

$$F(\vec{P}, \vec{\delta})\{N_{1c}\} \approx F(\vec{P})\{N_{1c}\} + \vec{J} * \vec{\delta}$$

donde:

$$\vec{J} = \frac{\partial F(\vec{P}, \vec{\delta})\{N_{1c}\}}{\partial \vec{P}}$$

, es el gradiente de la nube de puntos móvil transformada respecto a los parámetros de la transformación. Con esto podemos reescribir y reducir la ecuación a minimizar mediante los siguientes pasos:

$$S(\vec{P} + \delta) \approx \sum_{i=1}^{m} [N_{2i} - F(\vec{P})\{N_{1c}\} - \vec{J} * \vec{\delta}]^2 \approx \left\| N_2 - F(\vec{P})\{N_1\} - \vec{J} * \vec{\delta} \right\|^2$$

$$\approx [N_2 - F(\vec{P})\{N_1\}]^T * [N_2 - F(\vec{P})\{N_1\}] - 2 * [N_2 - F(\vec{P})\{N_1\}]^T * J * \delta + \delta^T * J^T * J * \delta$$

Para minimizar la ecuación anterior respecto al paso delta, la derivamos e igualamos a cero, llegándose a la condición:

$$(J * J^T) * \delta = J^T * [N_2 - F(\vec{P})\{N_{1c}\}]$$

La ecuación vectorial anterior, deriva en un sistema de ecuaciones lineales que se puede resolver respecto a delta. La diferencia de este algoritmo respecto a otros que resuelven cuadrados mínimos, es la introducción de un factor de amortiguamiento lambda:

$$(J * J^T + \lambda * I) * \delta = J^T * [N_2 - F(\vec{P})\{N_{1c}\}]$$

Si la minimización de S es veloz, se puede usar un lambda pequeño mientras que, si la convergencia al mínimo es lenta lambda puede aumentarse. Para establecer una condición de corte o convergencia se define que, si la magnitud del paso calculado delta, o la suma de delta con los parámetros, cae bajo un límite preestablecido, el último vector de P se toma como solución.

Se utilizó una metodología donde se comienza con un factor de amortiguamiento inicial y con un factor positivo "v". Luego, se calcula lambda con dos factores de amortiguación diferentes:

$$\lambda = \lambda_0$$
$$\lambda = \frac{\lambda_0}{v}$$

Si ejecutar una iteración con ambos factores empeora el valor de S, se incrementa el factor de convergencia por multiplicación sucesiva y los nuevos parámetros óptimos se calculan con este factor y el proceso sigue iterando:

$$\lambda = \lambda_0 * v^k$$

Si utilizar el factor corregido empeora el residuo, se debe analizar que sucede con el factor inicial. Si este último mejora el valor del residuo, se adopta como factor de amortiguamiento y se utiliza para calcular el nuevo valor de los parámetros óptimos.

Este optimizador se encuentra implementado en la clase de ITK "Levenberg-Marquadt Optimizer" y sus parámetros son altamente configurables.

8.9.4. Configuración del Optimizador.

Para controlar el optimizador de métrica de la registración, se utilizó el esquema "observer-command". En este esquema, cualquier tipo de vtkObject puede monitorearse en caso que cualquier evento lo invoque. Por ejemplo, el objeto vtkRenderer invoca un evento denominado "StartEvent" para comenzar el rendering en la ventana de visualización y un "EndEvent" cuando finaliza. Los observadores de evento se agregan a un objeto de vtk mediante el método "AddObserver()", requiriendo un evento (ya que se monitorea un evento particular en el objeto) y un "itkCommand" a ejecutarse en caso de que se registre un evento. En este marco, fue necesario seleccionar el tipo de evento a monitorear y fue útil generar una subclase a partir del comando.

Comenzamos por definir la subclase "CommandIterationUpdate" la cual hereda de itk::Command.

```
class CommandIterationUpdate : public itk::Command
{
public:
using Self = CommandIterationUpdate;
using Superclass = itk::Command;
using Pointer = itk::SmartPointer<Self>;
itkNewMacro( Self )
```

Definimos un constructor default protegido y, en cuanto sus atributos públicos, le asignamos el optimizador de tipo Levenberg Marquadt.

```
protected:
CommandIterationUpdate() = default;
public:
using OptimizerType = itk::LevenbergMarquardtOptimizer;
using OptimizerPointer= const OptimizerType *;
```

Luego haremos override sobre la función Execute de la clase itk::Command. Esta función, recibe un puntero al objeto que desencadena el evento (optimizador en este caso) y la dirección de memoria del evento propiamente dicho.

```
void Execute(itk::Object *caller, const itk::EventObject & event) override
{
Execute((const itk::Object *)caller, event);
}
void Execute(const itk::Object * object, const itk::EventObject & event) override
{
auto optimizer = dynamic_cast<OptimizerPointer>( object );
if(optimizer == nullptr )
{
itkExceptionMacro( "Could not cast optimizer." );
}
```

Notar que la función Execute recibe como primer parámetro un objeto de itk, por tanto es necesario castear el optimizador utilizando un dynamic cast. Luego nos resguardamos del evento null pointer:

```
if(! itk::IterationEvent().CheckEvent( &event ) )
{
return;
}
```

Por último, (continuamos dento de la función Execute) debemos indicar que un evento fuerce al optimizador a entregar sus valores actuales:

```
std::cout << "Value = " << optimizer->GetCachedValue() << std::endl;
std::cout << "Position = "  << optimizer->GetCachedCurrentPosition();
std::cout << std::endl << std::endl;
}
};
```

En síntesis, el script anterior crea la subclase "Command Iteration Event" a partir de la clase itkCommand. Para modificar la respuesta a un evento se overrideo la función Execute. Esta función ejecutará alguna rutina en base al objeto y evento asignado. De esta forma, se asignó como objeto al optimizador de la métrica. El evento que se monitorea es un evento de tipo iterativo. En caso de que el observador registre algún evento que rodea el objeto en cuestión, la función Execute obtendrá el valor y posición del optimizador y la devolverá en pantalla.

8.9.5. Consideraciones para gráficos 3D

Un problema que se presentó en este trabajo es la correcta integración e utilización complementaria de ITK y VTK. ¿Cómo aseguramos que los resultados obtenidos en una librería se adecúen a los de la otra? ¿Qué consideraciones habría que tener para, a partir de una registración computada en ITK, transformar y visualizar un PolyData en una ventana de VTK? Para resolver este tipo de problemas fue necesario explorar los manuales de ambos paquetes con el objetivo de determinar los sistemas de coordenadas y convenciones generales en el uso de gráficos computacionales. Es necesario por tanto establecer algunos conceptos.

Algunos paquetes que manipulan gráficos utilizan matrices de transformación "M" de 3x3 para transformar un punto o vector "v" con 3 coordenadas:

$$M = \begin{pmatrix} \mathrm{rs}_{11} & \mathrm{rs}_{12} & \mathrm{rs}_{13} \\ \mathrm{rs}_{21} & \mathrm{rs}_{22} & \mathrm{rs}_{23} \\ \mathrm{rs}_{31} & \mathrm{rs}_{32} & \mathrm{rs}_{33} \end{pmatrix} \tag{1}$$

Los elementos de dicha matriz, codifican el nivel de escalado y rotación en cada dirección y pueden multiplicar a un vector para rotarlo y escalarlo:

$$\vec{v} = \begin{pmatrix} \mathrm{v}_1 \\ \mathrm{v}_2 \\ \mathrm{v}_3 \end{pmatrix} \tag{2}$$

La operación que permite transformar dicho vector en un resultante "u" es entonces:

$$\vec{u} = \begin{pmatrix} \mathrm{rs}_{11} & \mathrm{rs}_{12} & \mathrm{rs}_{13} \\ \mathrm{rs}_{21} & \mathrm{rs}_{22} & \mathrm{rs}_{23} \\ \mathrm{rs}_{31} & \mathrm{rs}_{32} & \mathrm{rs}_{33} \end{pmatrix} . \begin{pmatrix} \mathrm{v}_1 \\ \mathrm{v}_2 \\ \mathrm{v}_3 \end{pmatrix} = M.\vec{v}$$

Ahora bien, las librerías que manipulan gráficos utilizan dos convenciones para guardar arreglos multidimensionales en memoria (RAM), **"Row-Major"** y **"Column-Major"**. En el primer caso, los elementos de un arreglo se guardan secuencialmente en memoria por fila, mientras que en el segundo caso esto se hace guardando por columna en memoria. Esto implica que todos los cálculos que involucran matrices, vectores y utilizan convención column-major, multiplican en orden derecha-izquierda (como es el caso de la ecuación anterior). Esto debe ser tenido en cuenta para evitar cometer errores en la integración de los frameworks.

Continuando, se pueden utilizar matrices 3x3 para determinar transformaciones de vectores de 3 coordenadas, sin embargo es común usar matrices y vectores/puntos 4D. Esto no significa que estemos trabajando en una hipergeometría en 4D, es solamente un truco: una matriz en 3x3 solo puede multiplicar un vector en 3 dimensiones y una matriz 4x4 solo un vector en 4D. Entonces si tenemos una matriz 4x4, ¿cómo transformamos nuestro vector a 4D?

La regla consta de extender el vector 3D con un 0 y el punto con un 1 en su última coordenada:

$$\vec{v}_{4D} = \begin{pmatrix} \mathrm{v}_1 & \mathrm{v}_2 & \mathrm{v}_3 & 0 \end{pmatrix}$$

$$\vec{P}_{4D} = \begin{pmatrix} \mathrm{p}_1 & \mathrm{p}_2 & \mathrm{p}_3 & 1 \end{pmatrix}$$

El pasaje a coordenadas en 4D es crucial. Mas aún, es conveniente utilizar matrices ya que nos permiten concatenar transformaciones sucesivas. De esta manera, se logra expresar una transformación global como el producto de transformaciones individuales:

$$\vec{v}_{transformado} = M_1.M_2.M_3...M_{n-2}.M_{n-1}.M_n$$

Luego, se debe mencionar que no se pueden representar traslaciones con matrices 3x3 y esto se puede demostrar de una manera informal y formal: tratemos de trasladar el origen de coordenadas (0,0,0) a otro punto (1,0,0) utilizando una matriz 3x3:

$$\begin{pmatrix} \mathrm{a}_{11} & \mathrm{a}_{12} & \mathrm{a}_{13} \\ \mathrm{a}_{21} & \mathrm{a}_{22} & \mathrm{a}_{23} \\ \mathrm{a}_{31} & \mathrm{a}_{32} & \mathrm{a}_{33} \end{pmatrix} . \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} ? \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Lo anterior no es posible. Una respuesta más formal está dada por el **Teorema de Descomposición en Valores Singulares (SVD)**, que nos dice que en 3 dimensiones podemos generar una matriz como combinación de rotaciones y escalamiento, sin mención de una traslación.

Al utilizar una matriz en 4 dimensiones, estamos trabajando con transformaciones afines basadas en combinaciones posibles de transformaciones lineales en 3D incluyendo, traslación, rotación y escalamiento. La matriz de transformación de dimensiones 4x4 es de la forma:

$$\begin{pmatrix} \mathrm{rs}_{11} & \mathrm{rs}_{12} & \mathrm{rs}_{13} & \mathrm{t}_{14} \\ \mathrm{rs}_{21} & \mathrm{rs}_{22} & \mathrm{rs}_{23} & \mathrm{t}_{24} \\ \mathrm{rs}_{31} & \mathrm{rs}_{32} & \mathrm{rs}_{33} & \mathrm{t}_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Lo anterior representa la matriz más genérica posible para una transformación afín que realiza rotación, escalamiento y traslación, en 3 dimensiones.

Es importante destacar que multiplicar dos matrices afines tiene como resultado otra afín:

$$\begin{pmatrix} a1 & b1 & c1 & d1x \\ d1 & e1 & f1 & d1y \\ g1 & h1 & i1 & d1z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} a2 & b2 & c2 & d2x \\ d2 & e2 & f2 & d2y \\ g2 & h2 & i2 & d2z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a3 & b3 & c3 & d1x + d2x \\ d3 & e3 & f3 & d1y + d2y \\ g3 & h2 & i3 & d1z + d2z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

De esta manera se pueden acumular transformaciones que realicen todas las operaciones (escalado, rotación y traslación) mediante multiplicaciones sucesivas, asegurando que la transformación se mantiene afín.

Habiendo introducido y explicado la existencia de la matriz de transformación 4x4 para operar sobre objetos en 3D, podemos decir (aceptar sin demostración) que para transformar un sistema Row-Major en uno Column-Major o viceversa, es necesario transponer la matriz de transformación:

$$M_{row-major} = M^{\intercal}_{column-major}$$

Tanto ITK como VTK (vtkMatrix4x4) utilizan un sistema row-major por lo cual no es necesario transponer la matriz de transformación en ITK para obtener su equivalente en VTK.

Otro problema a resolver es la equivalencia entre coordenadas: obtuvimos la matriz de transformación para VTK luego de haber registrado en ITK pero ésta no está en las coordenadas correctas.

En VTK existen tres sistemas de coordinadas aunque son dos los principales:

- **Coordenadas globales:** consiste en un sistema cartesiano en un espacio tridimensional. Cada punto está representado por el triplete (xi,yi,zi) en los ejes x,y,z. Este sistema se utiliza para especificar la posición de los datos en la ventana de visualización, así como también normales y vectores.

- **Coordenadas locales:** este sistema utiliza una combinación de coordenadas geométricas y topológicas. Las coordenadas topológicas se usan para identificar una célula o subcelula en la superficie/mallado o PolyData. Por otro lado, las coordenadas geométricas describen una posición determinada dentro de una célula.

  La coordenada topológica en el sistema local es un "Id" que refiere a una célula en el dato estructural. Si es una célula compuesta por subcelulas, se utiliza un "subId". La combinación de Id y subID indican una "celula primaria" de manera unívoca. Para especificar la posición dentro de una célula primaria, se utiliza un set de coordenadas geométricas (también llamadas paramétricas), ya que dependen de la dimensión y topología de la célula.

  Para el ejemplo que se muestra en la figura 33, consideremos la línea representada por un dato de tipo Polyline. Se puede especificar un punto en la línea indicando el Id de la célula, el sub-id para la célula primaria y las coordenadas paramétricas de la línea. Ya que la línea es unidimensional (aunque sus puntos estén en un sistema de coordenadas global 3D), su ecuación está parametrizada por un único parámetro que aquí llamaremos "r". Dicho parámetro varía entre (0,1) siendo el subId igual a "i":

  $$x(r) = (1 - r)x_i + rx_{i+1}$$

  Para un objeto en 3D, se requieren 3 parámetros (r,s,t) para describir la coordenada local de una célula, concluyendo que cada tipo de célula tiene su descripción paramétrica.

  Retomando el problema inicial: la matriz de transformación 4x4 obtenida en ITK y homologada para VTK, debe adecuarse a coordenadas globales. Para esto, se utiliza la clase VtkTransformPolyDataFilter.

La transformación en ITK se estimó utilizando como centro de rotación el centro de masa del objeto. Para poder tener en cuenta esto en la transformación de VTK, es necesario emplear la siguiente estrategia:

1. Trasladar el objeto a transformar con el vector (-Centro de masa X, -Centro de masa Y, -Centro de masa Z). Es decir, llevar el objeto a partir de su centro de masa, al origen de coordenadas global de VTK.

2. Aplicar la rotación necesaria sobre el objeto.

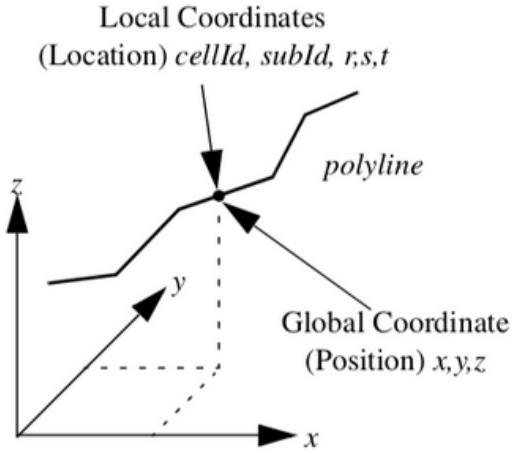


Figura 33: Sistema de coordenadas en VTK.

3. Volver a trasladar el objeto a su posición original.

Aquí terminamos con esta serie de consideraciones. En los incisos siguientes (particularmente en la sección número 8) utilizaremos estos conceptos de manera recurrente.

8.9.6. Selección de hiperparámetros

Al momento de registrar es necesario preconfigurar el algoritmo de registración. Los parámetros a configurar son los siguientes:

1. **Rotation scale**: una vez calculada la rotación necesaria para alinear los objetos en una iteración, es necesario desplazarnos hacia dicho valor. Esto se logra mediante un paso o escala de rotación. Seleccionar una escala muy grande podría producir máximos en el error de alineación. Por otro lado, si seleccionamos un paso pequeño y nos encontramos inicialmente en una posición desfavorable, corremos el riesgo de no escapar de un mínimo local.

2. **Translation scale**: mismo concepto que el de escala de rotación, pero aplicado a la traslación.

3. **Iteration number**: de no especificarse o cumplirse una condición de corte, se dejará de iterar según lo indique este campo.

4. **Gradient tolerance**: este es el valor a tolerar para el gradiente de la nube de puntos móvil transformada respecto a los parámetros de transformación. Es decir, detenemos la iteración cuando la "velocidad de movimiento" de la nube móvil es significativamente baja. Esto es notorio, cuando los valores de los parámetros en iteraciones sucesivas se vuelven similares.

5. **Value tolerance**: si la métrica de alineamiento cae debajo de este valor, se considera que ambos objetos están alineados.

6. **Epsilon value**: es la precisión de las operaciones que se realizan.

Ciertos valores se pueden dejar fijos, como por ejemplo el valor de Epsilon, o la tolerancia al valor deseado, así como también la tolerancia al gradiente y la cantidad de iteraciones. Para los parámetros remanentes, se puede conjeturar que, si se realiza un prealineamiento adecuado, podremos lograr una buena correspondencia. Esto sucede si los pasos son pequeños, es decir del orden de 0.001 para la traslación y 1 para la rotación. En cuanto a la cantidad de iteraciones, un número entre 500 y 1000 es adecuado para asegurar convergencia y disponibilizar resultados rápidamente. En la figura 34, se puede observar el panel de configuración del registrador.

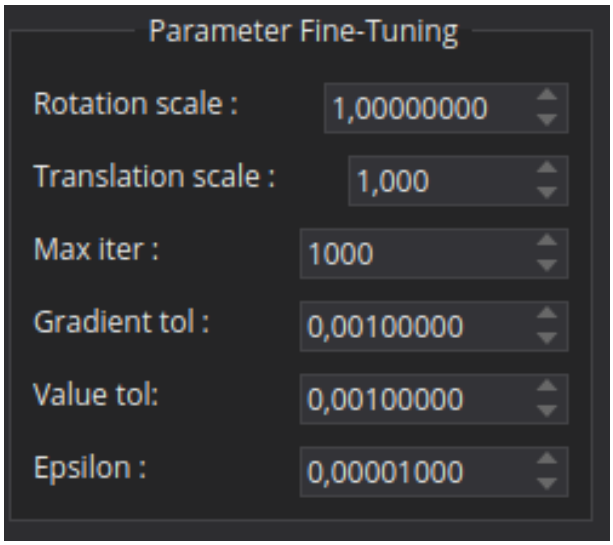


Figura 34: Panel de configuración del registrador

8.9.7. Funciones de configuración de la clase registerView

Antes de continuar, es necesario mostrar como fue implementada la clase RegisterView:

Primero, definiremos la función de la clase RegisterView que creará las vistas en la interfaz gráfica. Esta función se denomina "CreateQtPartControl". La misma recibe un objeto de la clase QWidget, que es la clase base para todos los objetos involucrados en la interfaz gráfica. Observar los siguientes extractos de código:

```
void registerView::CreateQtPartControl(QWidget* parent)
{

//Links interface with View
mControls.setupUi(parent);

//This allows linkage between the data storage and the target load field
mControls.target_box->SetDataStorage(this->GetDataStorage());

//Template box disabled until target is chosen
mControls.template_box_select->setEnabled(true);

//Connect storage to initialization box
mControls.initializing_box->SetDataStorage(this->GetDataStorage());
```

En el extracto anterior se vinculó la interfaz gráfica con la vista. Luego, los campos de ingreso para los objetos a registrar (fijo y móviles) se vinculan con el Data Storage. Lo mismo con el campo que permite inicializar cada nodo. Continuando:

```
//Setting predicate for target box
mControls.target_box->SetPredicate(mitk::NodePredicateAnd::New(
  mitk::TNodePredicateDataType<mitk::Surface>::New(),
  mitk::NodePredicateNot::New(mitk::NodePredicateOr::New(
    mitk::NodePredicateProperty::New("helper object"),
    mitk::NodePredicateProperty::New("hidden object")))));

//Setting predicate for initialization box
mControls.initializing_box->SetPredicate(mitk::NodePredicateAnd::New(
  mitk::TNodePredicateDataType<mitk::Surface>::New(),
  mitk::NodePredicateNot::New(mitk::NodePredicateOr::New(
    mitk::NodePredicateProperty::New("helper object"),
    mitk::NodePredicateProperty::New("hidden object")))));
```

En el extracto anterior se filtran los posibles inputs que toman los campos de inicialización de nodos que

provengan del Data Storage. Se filtra en particular que los mismos sean estructuras de superficie (mallado) de MITK.

Luego se realizan múltiples conexiones con la lógica signal-slot:

Cada vez que cambie el input del campo de inicialización de nodos o carga de nodos fijos o móviles, se ejecutará la función "OnSelectedSurfaceChanged":

```
connect(mControls.initializing_box, SIGNAL(OnSelectionChanged(const mitk::DataNode *)), this,
    SLOT(OnSelectedSurfaceChanged(const mitk::DataNode *)));

connect(mControls.target_box, SIGNAL(OnSelectionChanged(const mitk::DataNode *)), this, SLOT(
    OnSelectedSurfaceChanged(const mitk::DataNode *)));

connect(mControls.template_box_select, SIGNAL(OnSelectionChanged(const mitk::DataNode *)),
    this, SLOT(OnSelectedSurfaceChanged(const mitk::DataNode *)));
```

Luego vinculamos más botones con funciones especificas como, por ejemplo, el algoritmo principal, la carga de nodos en los campos de nodo target y template, la inicialización de nodos, la configuración de ventana y la creación del mapa de proximidad (será explicado en la sección número 8):

```
connect(mControls.button_main,SIGNAL(clicked()),this,SLOT(main_algorithm()));

connect(mControls.button_templateconfirm,SIGNAL(clicked()),this,SLOT(
    confirm_selected_templates()));

connect(mControls.button_targetconfirm,SIGNAL(clicked()),this,SLOT(confirm_selected_targets())
    );

connect(mControls.window_configure_button,SIGNAL(clicked()),this,SLOT(
    configure_window_interactor_renderer()));

connect(mControls.button_colormap,SIGNAL(clicked()),this,SLOT(compute_colormap()));

connect(mControls.button_initialize,SIGNAL(clicked()),this,SLOT(node_center()));
```

Conectamos los botones de movimiento con sus funciones de control correspondientes mediante la señal de click:

```
connect(mControls.alignment_help, SIGNAL(clicked(bool)), this, SLOT(UseTutorial(bool)) );

connect(mControls.control_up, SIGNAL(clicked(bool)), this, SLOT(control_up()) );

connect(mControls.control_down, SIGNAL(clicked(bool)), this, SLOT(control_down()));

connect(mControls.control_left, SIGNAL(clicked(bool)), this, SLOT(control_left()));

connect(mControls.control_right, SIGNAL(clicked(bool)), this, SLOT(control_right()));

connect(mControls.control_depth, SIGNAL(clicked(bool)), this, SLOT(control_depth()));

connect(mControls.control_shallow, SIGNAL(clicked(bool)), this, SLOT(control_shallow()));

connect(mControls.control_rx, SIGNAL(clicked(bool)), this, SLOT(control_rotate_x()));

connect(mControls.control_ry, SIGNAL(clicked(bool)), this, SLOT(control_rotate_y()));

connect(mControls.control_rz, SIGNAL(clicked(bool)), this, SLOT(control_rotate_z()));

connect(mControls.stop_alignment, SIGNAL(clicked(bool)), this, SLOT(stop_prealignment()));
```

A continuación, se implementa la función "OnSelectedSurfaceChanged". Ésta asegura que, cuando uno de los campos de ingreso modifique su estado, ésta se actualice. De esta manera, se deshabilitan los campos en caso de que estén vacíos, evitando un crash de la aplicación en caso de null pointer:

```
void registerView::OnSelectedSurfaceChanged(const mitk::DataNode *node)
{
```

```
  if (node != nullptr)
  {
      mControls.target_box->setEnabled(true);
      mControls.template_box_select->setEnabled(true);
      mControls.initializing_box->setEnabled(true);
  }
  else
  {
      mControls.target_box->setEnabled(false);
      mControls.template_box_select->setEnabled(false);
      mControls.initializing_box->setEnabled(false);
  }
}
```

Estas fueron las funciones de configuración más importantes para la clase. El resto de las funcionalidades pertenecientes a registerView fueron o serán explicadas en la sección 8.

8.9.8. Diseño de procesos

La funcionalidad de este plugin puede describirse mediante el diagrama de procesos de la figura 35:

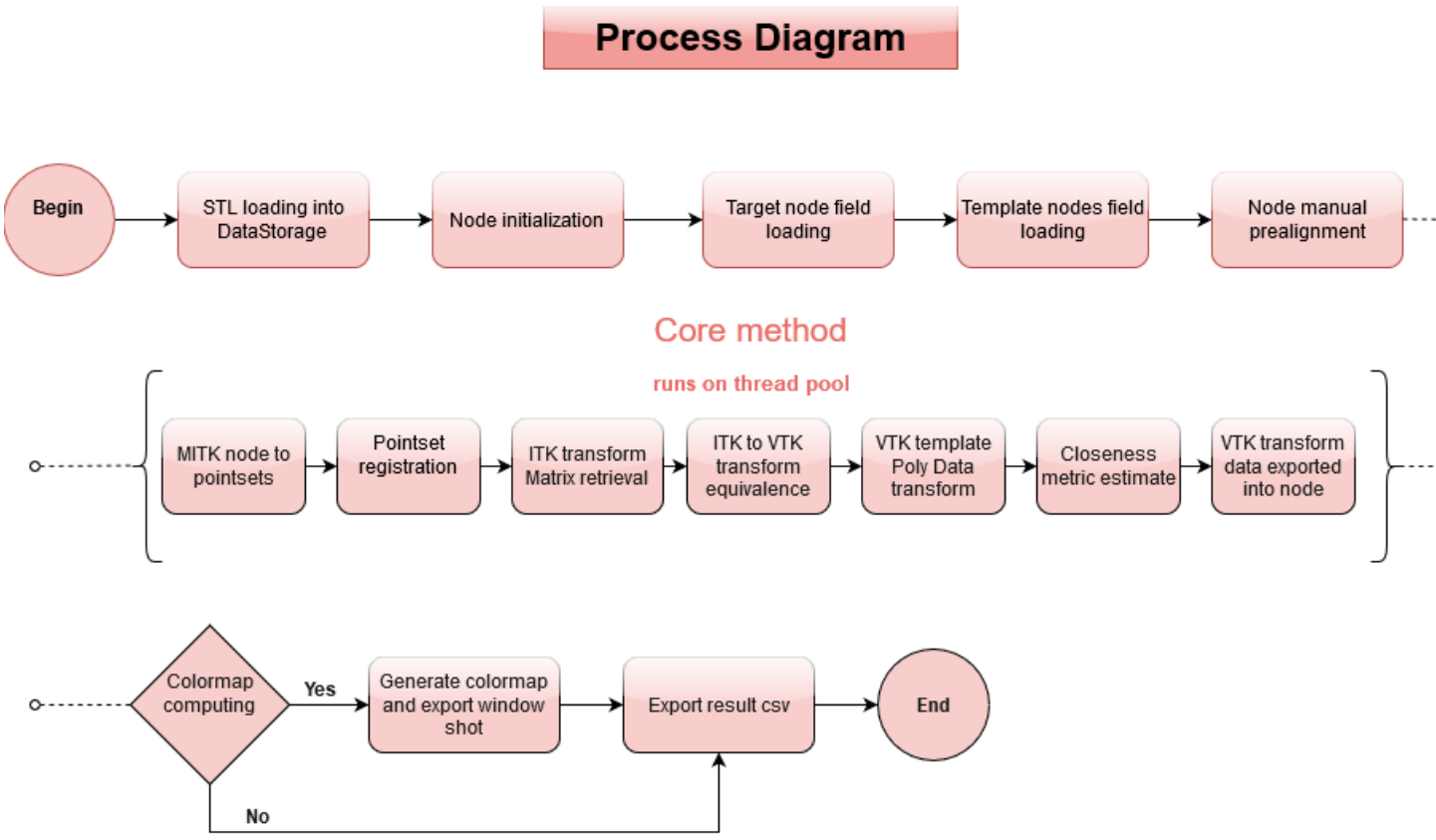


Figura 35: Diagrama de procesos para el plugin “register”.

1. El proceso comienza con la carga de todos los datos en formato STL (también usado como formato para imágenes o volúmenes en impresión 3D). Dichos datos se importan al Data Storage y figuran como nodos en el mismo.
2. Luego, se deben inicializar todos los nodos de manera secuencial centrándolos respecto al origen de coordenadas.

3. Se selecciona y carga el nodo fijo.
4. Se cargan todos los nodos móviles.
5. Se realiza el prealineamiento manual de los nodos móviles.
6. Se ejecuta el pipeline de registración
7. Según elección, puede computarse el mapa de proximidad. Por último, los resultados de la registración se exportan hacia un archivo con formato "csv".

En los siguientes extractos de código se muestra la implementación del Pipeline anterior:
El método registrador principal recibe un nodo móvil y se comunica con la interfaz para extraer el nodo fijo del Data Storage:

```
void registerView::DisplayOptimal(mitk::DataNode::Pointer template_node){
```

Luego obtenemos los parámetros de la interfaz:

```
double R_scale = mControls.r_scale_input->value();

double T_scale = mControls.t_scale_input->value();

unsigned int max_iter = (unsigned int)mControls.max_iter_input->value();

double grad_tol = mControls.grad_tol_input->value();

double value_tol = mControls.value_tol_input->value();

double epsilon = mControls.epsilon_input->value();
```

Luego, extraemos las superficies (mallados) de cada nodo (adicionalmente obtenemos el nodo target):

```
mitk::DataNode::Pointer target_node = mControls.target_box->GetSelectedNode();

mitk::Surface::ConstPointer target_surface = static_cast<mitk::Surface*>(target_node->GetData
    ());

mitk::Surface::ConstPointer template_surface = static_cast<mitk::Surface*>(template_node->
    GetData());
```

Luego, decimamos las superficies para obtener PolyDatas submuestreados:

```
vtkSmartPointer<vtkPolyData> target_polydata = downsample_surf(target_surface,20);

vtkSmartPointer<vtkPolyData> template_polydata = downsample_surf(template_surface,20);
```

Configuramos las nubes de puntos. Éstas tomarán puntos de ITK. Para esto definimos el objeto "Point Set" y creamos la nube de puntos móvil y fija:

```
typedef itk::PointSet<double,3> itkPointSetType;

itkPointSetType::Pointer itk_fixedSet = itkPointSetType::New();

itkPointSetType::Pointer itk_movingSet = itkPointSetType::New();
```

Luego, llenamos estas nubes con los puntos de los Poly Data decimados (se muestra solo el caso de la nube fija, la otra es análoga):

```
double point_retainer_1[3];

for(unsigned int i = 0; i<(target_polydata->GetNumberOfPoints()) ; i++){

target_polydata->GetPoint(i, point_retainer_1);

itk_fixedSet->SetPoint(i,point_retainer_1);
}
```

A partir de aquí configuramos la registración. Comenzamos definiendo y creando la métrica, que será la distancia euclídea:

```
typedef itk::Euler3DTransform<double> TransformType;

TransformType::Pointer Transform=TransformType::New();
```

Definimos una transformación basada en ángulos de Euler:

```
typedef itk::Euler3DTransform<double> TransformType;

TransformType::Pointer Transform=TransformType::New();
```

Se instancia el optimizador no-lineal paramétrico de Levenberg-Marquadt:

```
using OptimizerType = itk::LevenbergMarquardtOptimizer;

OptimizerType::Pointer optimizador = OptimizerType::New();

optimizador->SetUseCostFunctionGradient(false);
```

Se define el tipo de registración por puntos, recibiendo dos sets de puntos como inputs:

```
typedef itk::PointSetToPointSetRegistrationMethod<itkPointSetType,itkPointSetType>
    RegistrationType;
RegistrationType::Pointer registration=RegistrationType::New();
```

Se insertan los parámetros del optimizador:

```
optimizador->SetScales(scales);
optimizador->SetNumberOfIterations(numberOfIterations);
optimizador->SetValueTolerance(valueTolerance);
optimizador->SetGradientTolerance(gradientTolerance);
optimizador->SetEpsilonFunction(epsilonFunction);
```

Se configura la transformación. El centro de rotación elegido es el de cada objeto móvil:

```
vtkSmartPointer<vtkCenterOfm ss> template_m ss_center = vtkSmartPointer<vtkCenterOfm ss>::
    New();

template_m ss_center->SetInputData(template_polydata);
```

```
4
5  template_m ss_center->SetUseScalarsAsWeights(0);
6
7  template_m ss_center->Update();
8
9  double m ss_center_x =
10 (template_m ss_center->GetCenter())[0];
11
12 double m ss_center_y = (template_m ss_center->GetCenter())[1];
13
14 double m ss_center_z = (template_m ss_center->GetCenter())[2];
15
16 double centro[3]={m ss_center_x,m ss_center_y,m ss_center_z};
17
18 Transform->SetCenter(centro);
```

Los parámetros iniciales de rotación y desplazamiento serán 0, para evitar corrimientos respecto a la alineación manual:

```
1  double init_trans[3] = {0,0,0};
2
3  double init_rot[3] = {0,0,0};
4
5  Transform->SetRotation(init_rot[0],init_rot[1],init_rot[2]);
6
7  Transform->SetTranslation(init_trans);
8
9  registration->SetInitialTransformParameters(T
10 ransform->GetParameters());
```

Creamos un observador para el proceso de registración, el cual nos informará el valor de los parámetros y de la métrica en cada iteración:

```
1  ommandIterationUpdate::Pointer observer = CommandIterationUpdate::New();
2
3  optimizador->AddObserver( itk::IterationEvent(), observer );
```

Comienzan las iteraciones. Resguardamos la posibilidad de que haya algún error:

```
1  try
2  {
3  registration->Update();
4  }
5  catch( itk::ExceptionObject & e )
6  {
7  std::cerr << e << std::endl;
8  }
```

Aquí se deben destacar ciertos puntos. Primero, recordamos que la matriz de transformación y el vector de traslación fueron calculados a partir de las nubes de punto en ITK. Dos opciones se presentaron aquí:

- **A**: esta opción consiste en transformar la nube de puntos móvil con la matriz y el vector obtenido. Luego con esta nube de puntos ITK transformada se debe lograr un objeto superficie mediante la generación de un mallado triangular o cuadrangular cuyos vértices son los puntos de la nube. Luego, esta superficie podría ser insertada como un nodo de MITK al Data Storage, pudiendo ser manipulado por el usuario.
- **B**: esta segunda opción conlleva realizar un mapeo de la transformación obtenida en ITK hacia una transformación en VTK (con todos los recaudos que esto implica, explicados en secciones anteriores). Esto permitiría transformar de manera directa el Poly Data decimado, a partir del mismo extraer la superficie y agregarla como un nodo al Data Storage.

La opción más conveniente resultó ser la segunda ya que no debía disponer de un método de reconstrucción de mallado. De esta manera reconstruimos la matriz de transformación afín en VTK:

```
vtkSmartPointer<vtkMatrix4x4> Transform_Matrix = vtkSmartPointer<vtkMatrix4x4>::New();
Transform_Matrix->SetElement(0,0,(Transform->GetMatrix())[0][0]);
Transform_Matrix->SetElement(0,1,(Transform->GetMatrix())[0][1]);
Transform_Matrix->SetElement(0,2,(Transform->GetMatrix())[0][2]);
Transform_Matrix->SetElement(1,0,(Transform->GetMatrix())[1][0]);
Transform_Matrix->SetElement(1,1,(Transform->GetMatrix())[1][1]);
Transform_Matrix->SetElement(1,2,(Transform->GetMatrix())[1][2]);
Transform_Matrix->SetElement(2,0,(Transform->GetMatrix())[2][0]);
Transform_Matrix->SetElement(2,1,(Transform->GetMatrix())[2][1]);
Transform_Matrix->SetElement(2,2,(Transform->GetMatrix())[2][2]);

Transform_Matrix->SetElement(3,0,1);
Transform_Matrix->SetElement(3,1,1);
Transform_Matrix->SetElement(3,2,1);
Transform_Matrix->SetElement(3,3,1);

Transform_Matrix->SetElement(0,3,1);
Transform_Matrix->SetElement(1,3,1);
Transform_Matrix->SetElement(2,3,1);
```

Trasladamos el objeto afuera del centro:

```
vtkSmartPointer<vtkTransform> final_transform_first =
    vtkSmartPointer<vtkTransform>::New();

final_transform_first->Translate(-centro[0],-centro[1],-centro[2]);

vtkSmartPointer<vtkTransformPolyDataFilter> transformFilter_first =
    vtkSmartPointer<vtkTransformPolyDataFilter>::New();

transformFilter_first->SetTransform(final_transform_first);

transformFilter_first->SetInputData(template_polydata);

transformFilter_first->Update();
```

Rotamos:

```
vtkSmartPointer<vtkTransform> final_transform_second =
    vtkSmartPointer<vtkTransform>::New();

final_transform_second->SetMatrix(Transform_Matrix);

final_transform_second->Translate(0,0,0);

vtkSmartPointer<vtkTransformPolyDataFilter> transformFilter_second =
    vtkSmartPointer<vtkTransformPolyDataFilter>::New();

transformFilter_second->SetTransform(final_transform_second);

transformFilter_second->SetInputConnection(transformFilter_first->GetOutputPort());

transformFilter_second->Update();
```

Trasladamos el objeto de nuevo al centro de rotación original:

```
vtkSmartPointer<vtkTransform> final_transform_third =
    vtkSmartPointer<vtkTransform>::New();

```

```
final_transform_third->Translate(centro[0],centro[1],centro[2]);

vtkSmartPointer<vtkTransformPolyDataFilter> transformFilter_third =
    vtkSmartPointer<vtkTransformPolyDataFilter>::New();

transformFilter_third->SetTransform(final_transform_third);

transformFilter_third->SetInputConnection(transformFilter_second->GetOutputPort());

transformFilter_third->Update();
```

Aplicamos la traslación principal:

```
double Tx = (Transform->GetParameters())[3];
double Ty = (Transform->GetParameters())[4];
double Tz = (Transform->GetParameters())[5];

vtkSmartPointer<vtkTransform> final_transform_fourth =
    vtkSmartPointer<vtkTransform>::New();

final_transform_fourth->Translate(Tx,Ty,Tz);

vtkSmartPointer<vtkTransformPolyDataFilter> transformFilter_fourth =
    vtkSmartPointer<vtkTransformPolyDataFilter>::New();

transformFilter_fourth->SetTransform(final_transform_fourth);

transformFilter_fourth->SetInputConnection(transformFilter_third->GetOutputPort());

transformFilter_fourth->Update();
```

Obtenemos el output de estas transformaciones sucesivas:

```
vtkSmartPointer<vtkPolyData> polydata_registrado=transformFilter_fourth->GetOutput();
```

Configuramos la superficie registrada:

```
mitk::Surface::Pointer polydata_registered_surf = mitk::Surface::New();
polydata_registered_surf->SetVtkPolyData(polydata_registrado);
```

Configuramos e insertamos el nodo en el Data Storage:

```
mitk::DataNode::Pointer node_registered_polydata = mitk::DataNode::New();

node_registered_polydata->SetData(polydata_registered_surf);

node_registered_polydata->SetName(template_node->GetName() + "_registered");

node_registered_polydata->SetColor(0.1,0,0.3);

GetDataStorage()->Add(node_registered_polydata);
```

Por último, el calculo de la métrica consiste en el valor Root Mean Square de la diferencia entre el objeto fijo y el objeto móvil. Dicho calculo se encuentra implementado en la función de generación de mapa de proximidad, descripta en la sección 8.10.1.

8.9.9. Método principal

La función de ejecución principal se encarga de ejecutar la registración de manera paralela y eficiente sobre todos los nodos seleccionados. Si bien la velocidad de procesamiento depende de cada ordenador, no hace falta resguardarse con el tamaño del input ya que la Pool de Threads se asegura que, cuando uno de estos termina, es asignado inmediatamente a otra registración. A continuación se muestra la implementación:

Primero caracterizamos el thread principal (esto no es necesario, es solamente por una cuestión de organización):

```
void registerView::main_algorithm(){

//Main thread
QThread::currentThread()->setObjectName("Main Thread");
std::cout<<"Multithreading is being called from: "<<QThread::currentThreadId()<<std::endl;
```

Luego seleccionamos el nodo fijo o target:

```
// Initializing target node
mitk::DataNode::Pointer target_node = mControls.target_box->GetSelectedNode();
```

Seleccionamos todos los nodos móviles:

```
mitk::DataStorage::SetOfObjects::ConstPointer rs = mControls.template_box_select->GetNodes();
```

Con dichos nodos:

```
for (unsigned int i = 0; i < rs->size(); i++){
///// Operations on nodes //////
}
```

En las iteraciones, primero debemos asegurar no registrar un nodo consigo mismo:

```
mitk::DataNode::Pointer template_node=rs->GetElement(i);
if (template_node->GetName() == target_node->GetName() ){
std::cout<<"Not registering object with itself"<<std::endl;
```

Para el resto de los nodos ejecutaremos la registración de manera paralela, como fue explicado en la sección 5.7:

```
QtConcurrent::run(this,&registerView::DisplayOptimal,template_node);
```

Por último, es importante destacar una funcionalidad adicional que se intentó implementar, pero que no pudo concretarse. Se trata de un monitoreo del progreso de la registración sobre los threads, de manera de poder supervisar el progreso de cada registración. Esto se quiso implementar mediante un QWatcher sobre la función a monitorear:

Primero, se crea un QProgressDialog que mostrará el progreso de la registración en porcentaje para cada thread:

```
QProgressDialog dialog;

dialog.setLabelText(QString("Progressing using %1 thread(s)...").arg(QThread::idealThreadCount
    ()))
```

Luego se crea el objeto que realiza el monitoreo del proceso paralelo. En este caso, no retornará ningún objeto ya que la registración exportará los nodos al Data Storage de manera directa:

```
QFutureWatcher<void> futureWatcher;
```

Luego realizamos múltiples conexiones. Primero especificaremos que, en caso de que el monitoreo termine para un thread, la barra de progreso se debe reiniciar.

```
QObject::connect(&futureWatcher, SIGNAL(finished()), &dialog, SLOT(reset()));
```

Si se cancelara la barra de progreso, el monitoreo terminaría:

```
QObject::connect(&dialog, SIGNAL(canceled()), &futureWatcher, SLOT(cancel()));
```

Cuando el monitoreo registre un avance temporal, la barra de progreso se actualizará.

```
QObject::connect(&futureWatcher, SIGNAL(progressValueChanged(int)), &dialog,SLOT(setValue(int)
    ));
```

Luego, se ejecutará el proceso de registración en paralelo:

```
QFuture<void> future = QtConcurrent::run(this,&registerView::DisplayOptimal, template_node);
```

Luego, se vincula el monitor de resultados con el procesamiento en paralelo, se inicializa la barra de progreso y se espera a que este termine:

```
futureWatcher.setFuture(future);

dialog.exec();

futureWatcher.waitForFinished();
```

El approach propuesto en los extractos anteriores dio buenos resultados en una instancia inicial, pero falló posteriormente. Esto se debe a que, en instancias más avanzadas del proyecto, fue necesario que cada registración tenga acceso directo a la interfaz gráfica. Esto significa que cada hilo debía interactuar con un objeto de la clase RegisterView (interfaz mControls) que se encuentra en el thread principal, separado de los threads secundarios de registración. Resolver esto tiene una complejidad que, prolongaría el tiempo de realización de este proyecto, ya que se deberían vincular distintos threads con el thread principal y que éstos no generen overlapping al demandar información de la interfaz. Por estas razones, el monitoreo del progreso se dejó a un lado, debido además a que la duración de la registración es suficientemente baja y que el usuario puede seguir utilizando la interfaz mientras esto sucede.

8.10. Medición de proximidad.

Se propuso una herramienta para cuantificar la cercanía entre los objetos registrados. Para lograr esto, se utilizó una medición de proximidad basada en la distancia euclídea. Se computa la distancia, o error de proximidad, entre cada punto “i” en la nube de puntos N1, con su correspondiente “j” en la nube N2, según:

$$E(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$$

Para poder visualizarlo de manera directa, se creó un mapa de proximidad (también llamado mapa de color) con una correspondencia entre color y proximidad (representada por la ecuación anterior). Dicho valor se calcula en coordenadas globales.

Por otro lado, para poder cuantificar el error global en la alineación lograda, se promedió el valor el error de proximidad para todos los puntos alineados. Adoptaremos entonces la siguiente ecuación como el **Error de Registración Global** al alinear un objeto A con otro B:

$$EG(A,B) = \sum_{i=1}^{N_{min}} E(i_A, i_B)/N_{min}$$

, donde N_{min} representa la cantidad de puntos en el objeto de menor tamaño.

8.10.1. Consideraciones sobre las métricas calculadas.

Las métricas de proximidad calculadas anteriormente, se obtienen en coordenadas globales. Por otro lado, MITK trabaja con el formato DICOM. En dicho estándar, se encuentra toda la información del dispositivo que genera las imágenes. De esta manera, al transformar las imágenes en un volumen con formato STL, también se conserva la información de las escalas, el espaciado y la dirección. Con esto se construyen los PolyDatas que se manipulan en las ventanas de MITK. En concreto, esto significa que las coordenadas globales están expresadas en milímetros. Sin embargo, para acelerar el proceso de registración, se realizó una decimación de los PolyData. Esto hace que se pierdan vértices en los objetos y, por tanto, que se pierda el mapeo 1 a 1 entre coordenadas globales y milímetro. Por esto, se tuvieron en cuenta dos parámetros mencionados en la sección 8.7:

- **Optimization level**: este factor se configuró en 1. De esta manera, se minimizó la distancia entre la superficie original y la decimada.
- **Boundary fixing**: este parámetro se configuró en "True". De esta manera, se realiza un padding con vértices extra, para mantener los límites de las superficies originales.

Con estas dos consideraciones, podemos asegurar que tenemos una aproximación adecuada para la correspondencia entre coordenadas globales y milímetros.

Por último, es importante destacar que, el error de registración global que califica la registración obtenida, es una estimación. Esto se debe a que, dicho error, es calculado como la suma de distancias entre puntos coincidentes. Al trabajar con huesos diferentes o, al intentar encontrar coincidencia entre subregiones de los mismos, se comete un error. Por esto, el valor calculado debe tomarse como una estimación.

8.10.2. Implementación de un mapa de proximidad.

Este mapa de color o proximidad, se obtiene mediante el cálculo de la distancia entre un objeto fijo y el móvil. La medida tomada es la distancia no signada desde cada punto del objeto fijo al punto más próximo en el móvil. Luego de obtener dichos valores, se genera un mapeo de dicha métrica a un color sobre el objeto fijo. El cálculo y generación del mapa de color se muestra en la siguiente función, cuyos parámetros de entrada son los nodos del Data Manager correspondiente a los objetos a comparar:

```
void registerView::calculate_distance(mitk::DataNode::Pointer fixed,
mitk::DataNode::Pointer moving){
```

Extraemos los Poly Datas de los nodos:

```

mitk::Surface::ConstPointer fixed_surface = static_cast<mitk::Surface*>(fixed->GetData());
vtkSmartPointer<vtkPolyData> fixed_polydata = fixed_surface->GetVtkPolyData();

mitk::Surface::ConstPointer moving_surface = static_cast<mitk::Surface*>(moving->GetData());
vtkSmartPointer<vtkPolyData> moving_polydata = moving_surface->GetVtkPolyData();
```

Luego, aplicamos un filtrado sobre los PolyDatas, entre otras utilidades removiendo puntos duplicados:

```
vtkSmartPointer<vtkCleanPolyData> clean1 =
  vtkSmartPointer<vtkCleanPolyData>::New();
clean1->SetInputData( fixed_polydata);

vtkSmartPointer<vtkCleanPolyData> clean2 =
  vtkSmartPointer<vtkCleanPolyData>::New();
clean2->SetInputData(moving_polydata);
```

Luego, creamos un filtro de distancia y forzamos la magnitud a modo no signado. Luego conectamos los objetos a comparar:

```
vtkSmartPointer<vtkDistancePolyDataFilter> distanceFilter = vtkSmartPointer<
    vtkDistancePolyDataFilter>::New();distanceFilter->SignedDistanceOff();
distanceFilter->SignedDistanceOff();
distanceFilter->SetInputConnection(0, clean1->GetOutputPort());
distanceFilter->SetInputConnection(1,clean2->GetOutputPort());
distanceFilter->Update();
```

La salida de este filtro se conecta en un mapper que insertaremos en un actor de VTK. Luego obtendremos los valores extremos de las distancias calculadas y asignaremos los valores de distancia a un actor en forma de barra graduada:

```
vtkSmartPointer<vtkPolyDataMapper> mapper = vtkSmartPointer<vtkPolyDataMapper>::New();
mapper->SetInputConnection(distanceFilter->GetOutputPort());
mapper->SetScalarRange(
  distanceFilter->GetOutput()->GetPointData()->GetScalars()->GetRange()[0],
  distanceFilter->GetOutput()->GetPointData()->GetScalars()->GetRange()[1]);

vtkSmartPointer<vtkActor> distance_actor =
  vtkSmartPointer<vtkActor>::New();
distance_actor->SetMapper( mapper );

vtkSmartPointer<vtkScalarBarActor> scalarBar =
        vtkSmartPointer<vtkScalarBarActor>::New();

scalarBar->SetLookupTable(mapper->GetLookupTable());
scalarBar->SetTitle("Distance value");
scalarBar->SetNumberOfLabels(4);
```

Realizamos el renderizado de los actores sobre la ventana de registerView. Por esto, la función de cálculo de distancia debe heredar de la clase registerView:

```
mitk::BaseRenderer::Pointer threeD_renderer = GetRenderWindowPart()->GetQmitkRenderWindow("3d
    ")->GetRenderer();
threeD_renderer->GetVtkRenderer()->AddActor2D(scalarBar);
threeD_renderer->GetVtkRenderer()->AddActor(distance_actor);
}
```

Definiremos otra función que será ejecutada cuando se haga click en el boton correspondiente de la GUI. Previamente, generaremos una conexión en registerView (donde mControls representa la GUI):

```
connect(mControls.pushButton_colormap,SIGNAL(clicked()),this,SLOT(compute_colormap()));
```

Por último, definimos efectivamente la función que ejecutará el proceso anterior:

```
void registerView::compute_colormap(){
    mitk::DataNode::Pointer fixed_node = mControls.target_box->GetSelectedNode();
    mitk::DataNode::Pointer moving_node = mControls.template_box_select->GetSelectedNode();
    calculate_distance(fixed_node,moving_node);
}
```

En la figura número 36 se puede observar un ejemplo de mapa de proximidad.

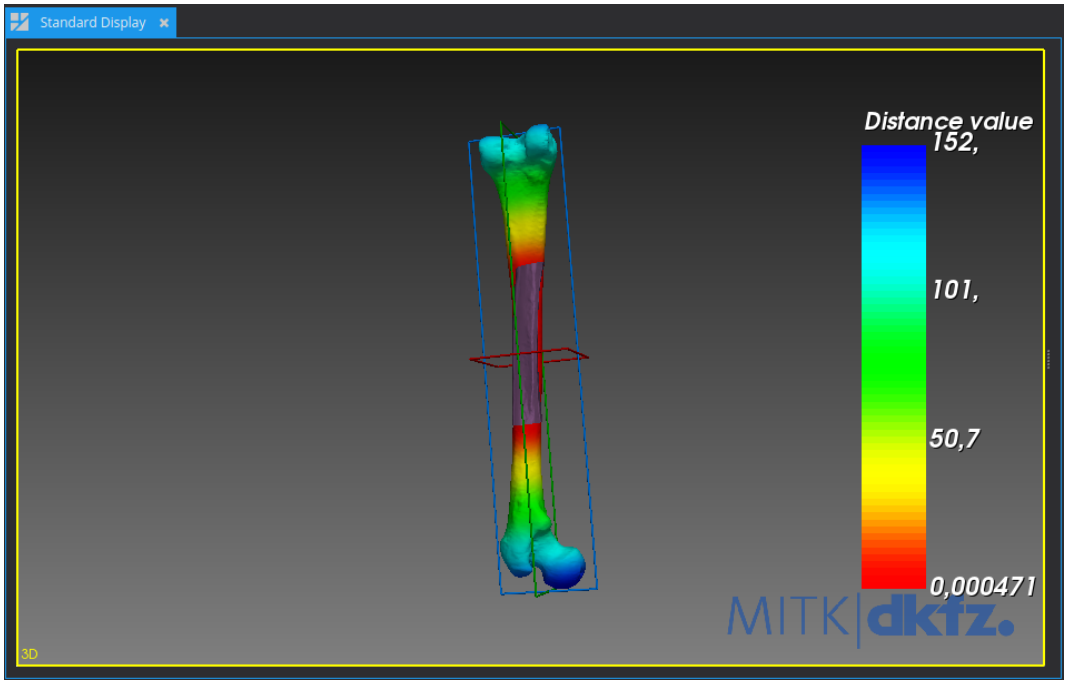


Figura 36: Mapa de proximidad implementado sobre el objeto fijo. También es posible invertir el mappeo. En este caso particular, la escala muestra valores de proximidad elevados. Esto se debe a que, al computar la distancia, se consideró el fémur como elemento principal.

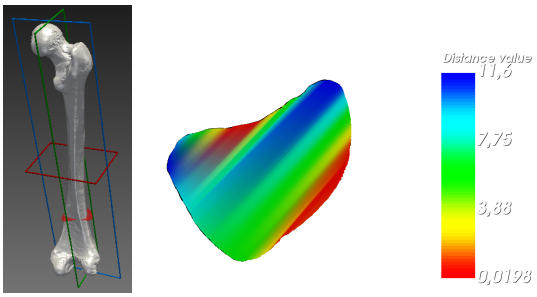
9. Resultados

9.1. Registración de huesos.

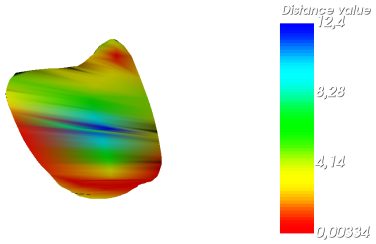
Para realizar pruebas de alineamiento, se seleccionaron 9 piezas de huesos correspondientes a muestras del Hospital Italiano de Buenos Aires:

1. Pieza 0: Fémur.
2. Pieza 1: Metáfisis.
3. Pieza 2: Fémur distal.
4. Pieza 3: Diáfisis.
5. Pieza 4: Cóndilo medial.
6. Pieza 5: Cóndilo lateral.
7. Pieza 6: Fémur.
8. Pieza 7: Fémur distal.
9. Pieza 8: Fémur distal.

Con estas 9 muestras se diseñaron múltiples pruebas independientes, donde la prueba (i, [j, k, ..., n]) representa la alineación de la pieza "i" con las piezas del arreglo [j, k, ..., n]. Para lograr una exposición ordenada de los resultados, se presentarán los mapas de proximidad iniciales (sub-figuras (b)) y finales (sub-figuras (c)), junto con el alineamiento inicial (sub-figuras (a)), en forma de mosaico. Luego se realizará una tabla global para todas las pruebas, que mostrará las métricas obtenidas. La ejecución de las pruebas se muestra en las siguientes imágenes:

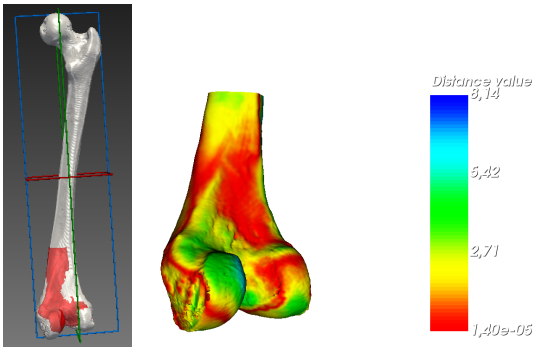


(a) (b)

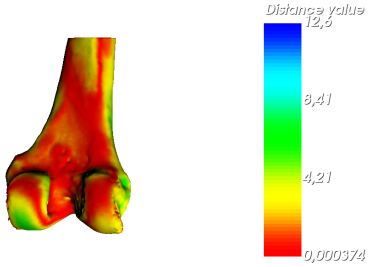


(c)

Figura 37: Prueba (0,1). Alineación global inicial de 6,493 milímetros. Alineación final de 3,713 milímetros. Mejora del 42,82 porciento en la alineación.



(a) (b)



(c)

Figura 38: Prueba (0,2). Alineación global inicial de 3,366 milímetros. Alineación final de 2,505 milímetros. Mejora global del 25,58 porciento.

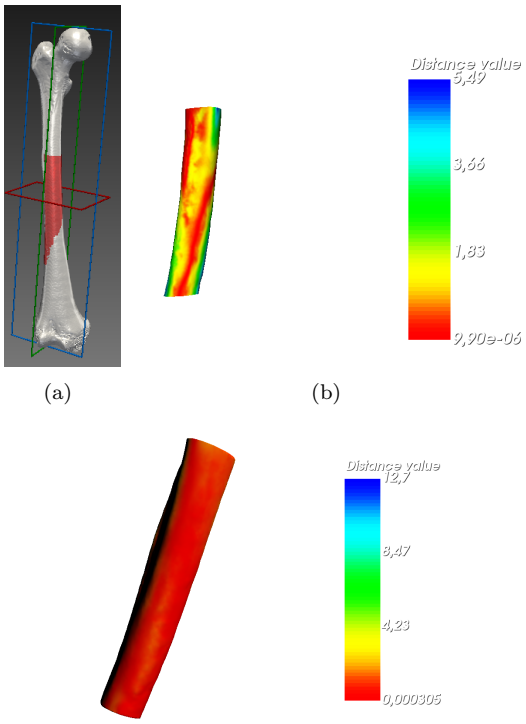
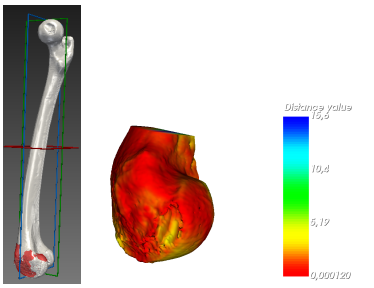
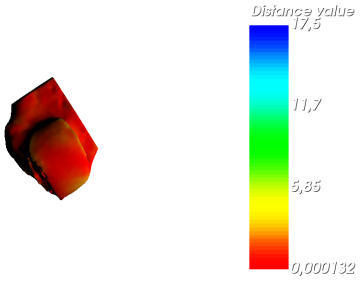


Figura 39: Prueba (0,3). Alineación global inicial de 2,286 milímetros. Alineación final de 1,729 milímetros. Mejora porcentual del 24,37 porciento.

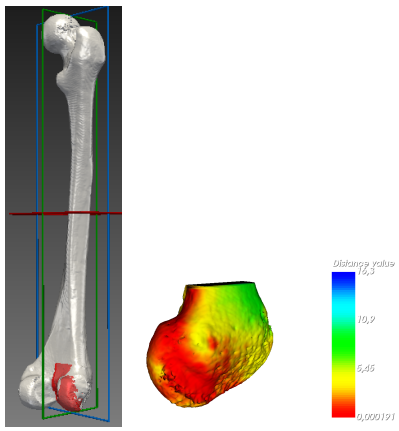


(a) (b)

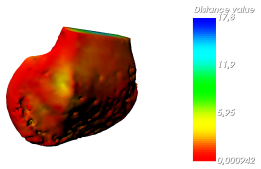


(c)

Figura 40: Prueba (0,4). Alineación global inicial de 1,889 milímetros. Alineación final de 2,490 milímetros. Disminución del 31,82 porciento en la alineación.

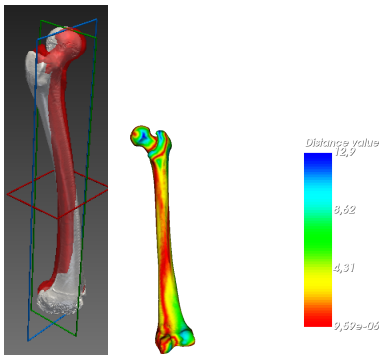


(a) (b)

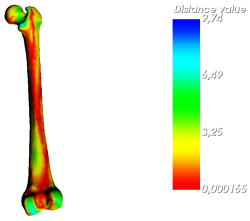


(c)

Figura 41: Prueba (0,5). Alineación global inicial de 3,548 milímetros. Alineación final de 2,759 milímetros. Mejora del 22,24 porciento en la alineación.

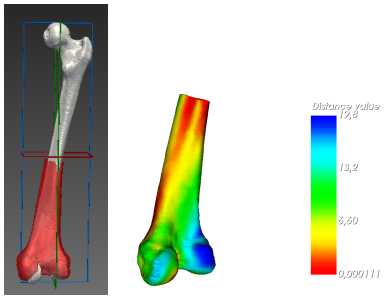


(a) (b)

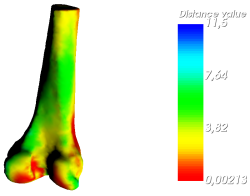


(c)

Figura 42: Prueba (0,6). Alineación global inicial de 4,337 milímetros. Alineación final de 2,486 milímetros. Mejora del 42,68 porciento en la alineación.

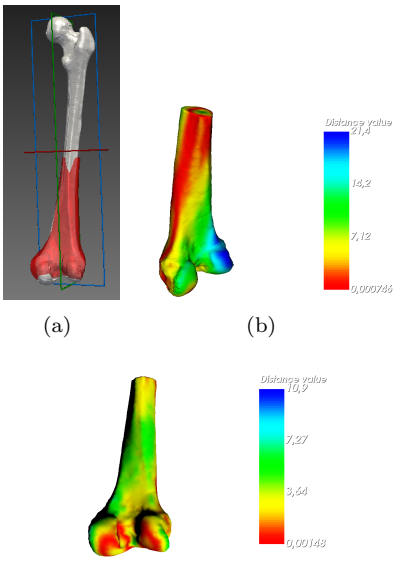


(a) (b)



(c)

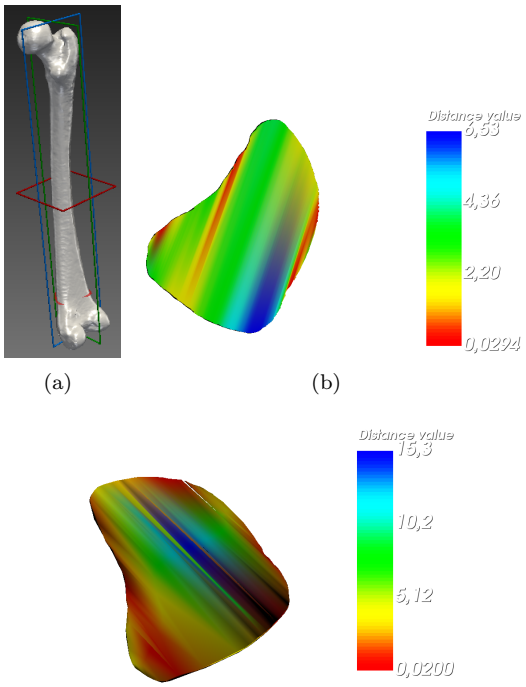
Figura 43: Prueba (0,7). Alineación global inicial de 7,205 milímetros. Alineación final de 11,459 milímetros. Disminución del 59,04 porciento en la alineación.



(a) (b)

(c)

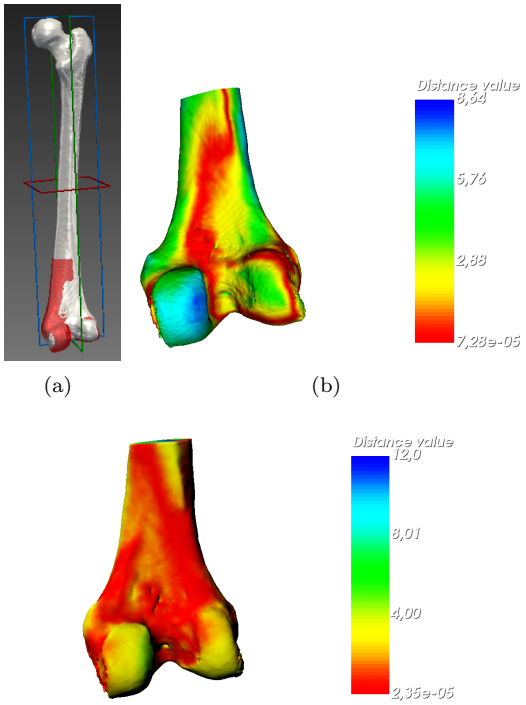
Figura 44: Prueba (0,8). Alineación global inicial de 7,322 milímetros. Alineación final de 3,679 milímetros. Mejora del 49,75 porciento en la alineación.



(a) (b)

(c)

Figura 45: Prueba (6,1). Alineación global inicial de 2,966 milímetros. Alineación final de 5,044 milímetros. Disminución del 70,06 porciento en la alineación.



(a) (b)

(c)

Figura 46: Prueba (6,2). Alineación global inicial de 3,579 milímetros. Alineación final de 2,631 milímetros. Mejora del 26,49 porciento en la alineación.

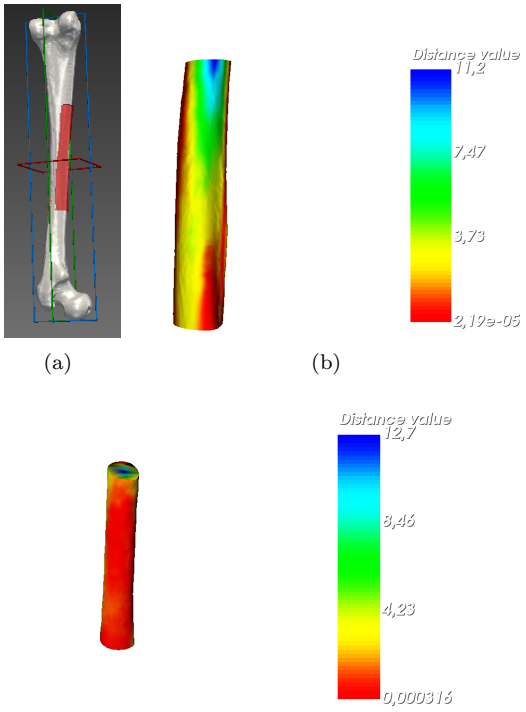


Figura 47: Prueba (6,3). Alineación global inicial de 2,905 milímetros. Alineación final de 2,118 milímetros. Mejora del 27,09 porciento en la alineación.

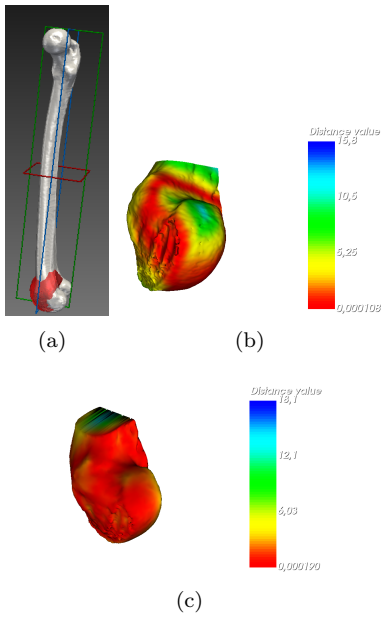


Figura 48: Prueba (6,4). Alineación global inicial de 4,615 milímetros. Alineación final de 2,582 milímetros. Mejora del 44,05 porciento en la alineación.

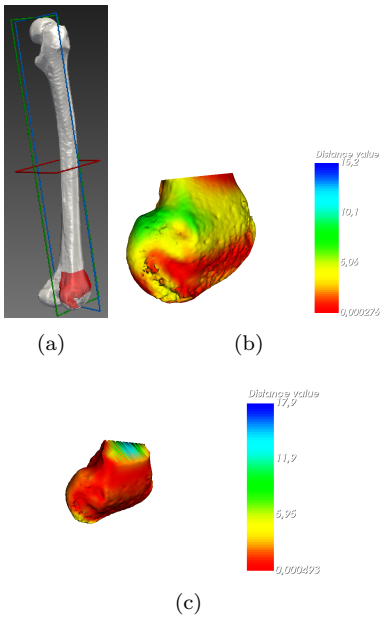


Figura 49: Prueba (6,5). Alineación global inicial de 3,634 milímetros. Alineación final de 2,404 milímetros. Mejora del 33,85 porciento en la alineación.

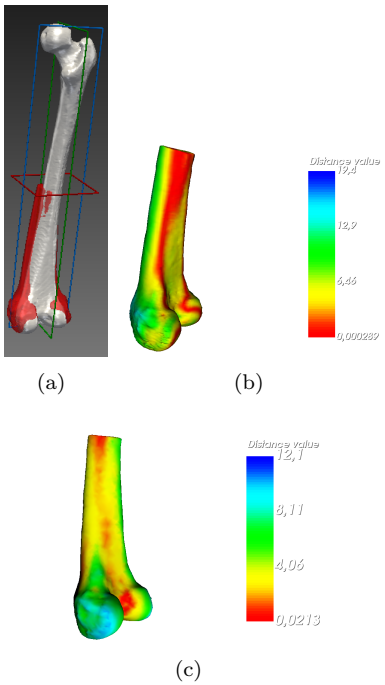
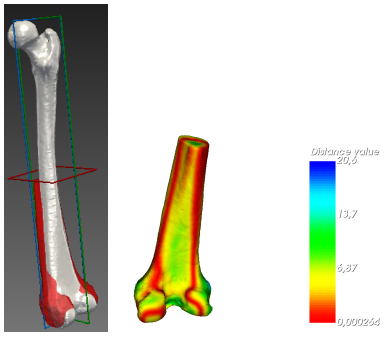
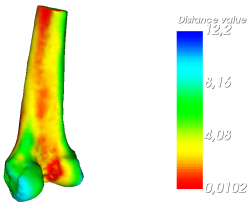


Figura 50: Prueba (6,7). Alineación global inicial de 8,403 milímetros. Alineación final de 5,538 milímetros. Mejora del 34,09 porciento en la alineación.

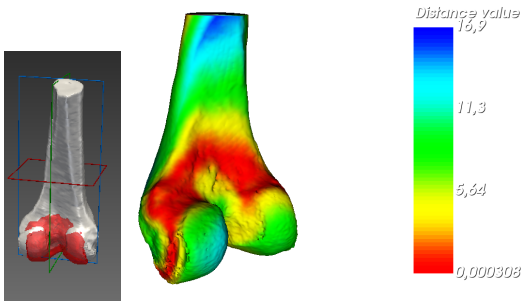


(a) (b)

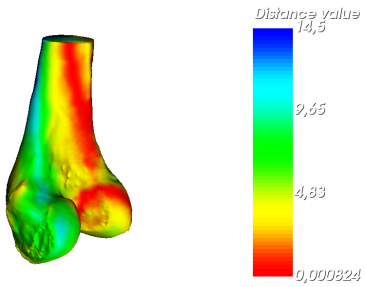


(c)

Figura 51: Prueba (6,8). Alineación global inicial de 8,059 milímetros. Alineación final de 5,123 milímetros. Mejora del 36,43 porciento en la alineación.

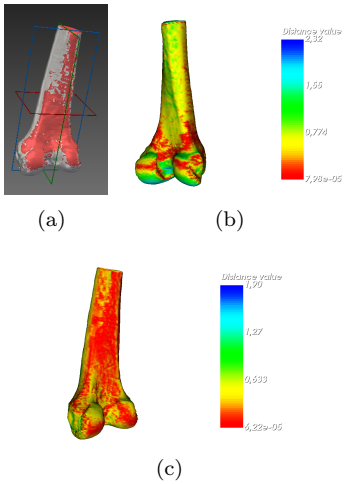


(a) (b)



(c)

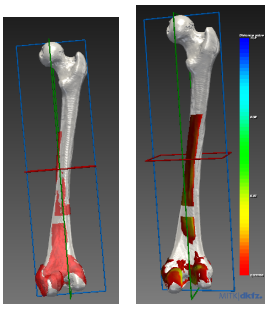
Figura 52: Prueba (7,2). Alineación global inicial de 7,765 milímetros. Alineación final de 0,706 milímetros. Mejora del 90,91 porciento en la alineación.



(a) (b)

(c)

Figura 53: Prueba (7,8). Alineación global inicial de 7,513 milímetros. Alineación final de 5,253 milímetros. Mejora del 30,08 porciento en la alineación.



(a) (b)

Figura 54: Prueba (0, [2,3,4,5]). Caso particular en donde se registraron múltiples objetos en simultaneo. Se muestran solo los objetos registrados a modo de ejemplificación.

Se calculó el error de registración global (EG), antes y después de la alineación. Los resultados se exhiben siguiente tabla:

Prueba (i,j)	EG - inicial [mm]	EG - final [mm]
(0,1)	6.493	3.713
(0,2)	3.366	2.505
(0,3)	2.286	1.729
(0,4)	1.889	2.490
(0,5)	3.548	2.759
(0,6)	4.337	2.486
(0,7)	7.205	11.459
(0,8)	7.322	3.679
(6,1)	2.966	5.044
(6,2)	3.579	2.631
(6,3)	2.905	2.118
(6,4)	4.615	2.582
(6,5)	3.634	2.404
(6,7)	8.403	5.538
(6,8)	8.059	5.123
(7,2)	7.765	0.706
(7,8)	7.513	5.253
(0, [2,3,4,5])	—	(2.512,1.727,2.458,2.773)

Los resultados de las pruebas muestran, en promedio, un aumento en la alineación arriba del 21,47 %. Las pruebas que arrojan resultados negativos se deben a muestras muy grandes comparadas con la muestra fija, por lo cual el algoritmo tiende a descolocarla aún más. Este último caso se resuelve tomando simplemente la estimación inicial, dada por el prealineamiento. Si removemos estos 3 valores negativos, correspondientes a las pruebas (0,4), (0,7) y (6,1), tendríamos un aumento en la proximidad post registración de 37,89 %. Esto corresponde a un aumento promedio de 2.185 milímetros en la alineación. Por último, en el caso de las pruebas (0,1) y (6,1), debemos mencionar la dificultar de alinear el fémur con la metáfisis debido a que el radio externo de ésta puede superar el del fémur y, por tanto, el algoritmo resuelve inclinando el disco y posicionándolo de manera incorrecta. Se podría resolver esto si se desarrollase una métrica de similitud de radio, aunque esto está fuera del alcance del proyecto, ya que solo nos concentraríamos en huesos largos.

9.2. Generación de cortes bidimensionales

En este inciso se muestran los resultados en la generación de cortes bidimensionales. La presentación de los resultados incluye, vista axial, vista sagital, vista coronal y vista tridimensional de los objetos que se ejemplifican. Cada corte esta representado por un color:

1. Corte sagital: demarcado en verde.
2. Corte axial: demarcado en rojo.
3. Corte coronal: demarcado en azul.
4. Ventana 3-D: demarcado en amarillo.

Observar dichas vistas en las siguientes imágenes (55, 56, 57, 58, 59 y 60):

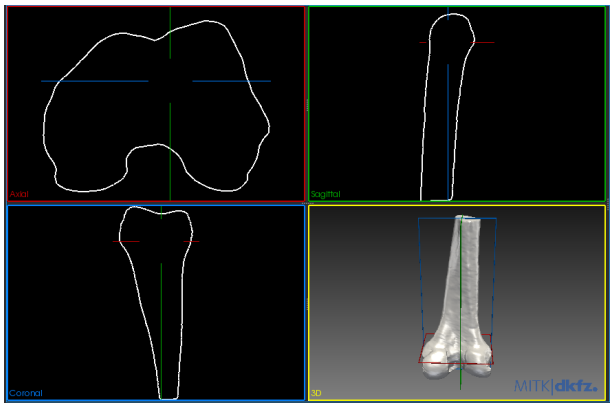


Figura 55: Se pueden observar las vistas 2-D de una porción de fémur inferior.

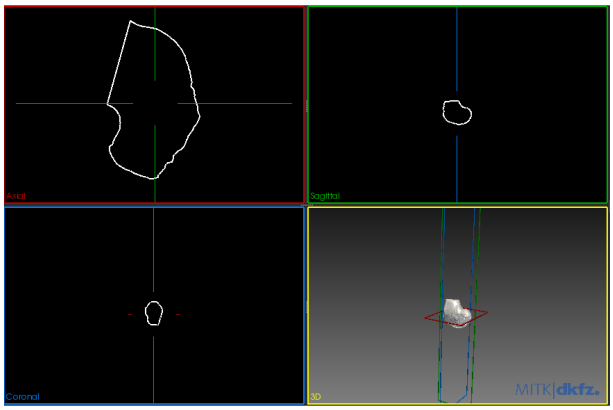


Figura 56: Se pueden observar las vistas 2-D de un cóndilo lateral.

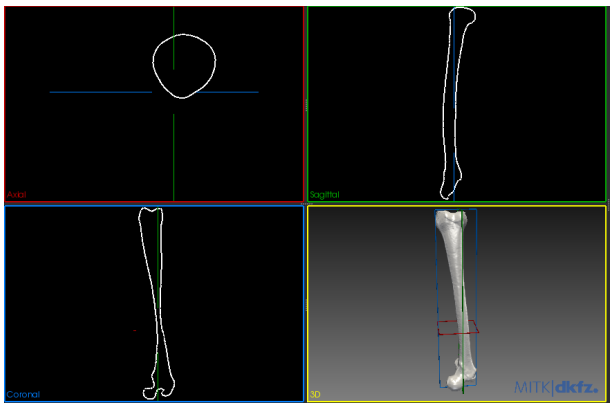


Figura 57: Se pueden observar las vistas 2-D de un fémur completo.

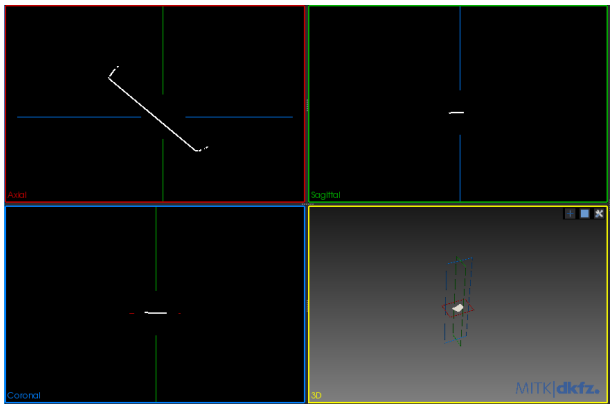


Figura 58: Se pueden observar las vistas 2-D de una sección de metáfisis femoral.

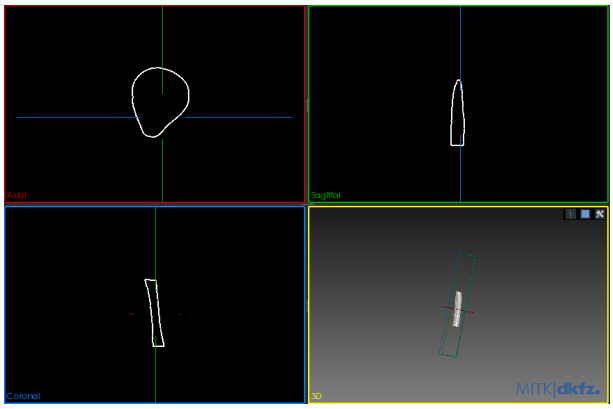


Figura 59: Se pueden observar las vistas 2-D de una sección de diáfisis femoral.

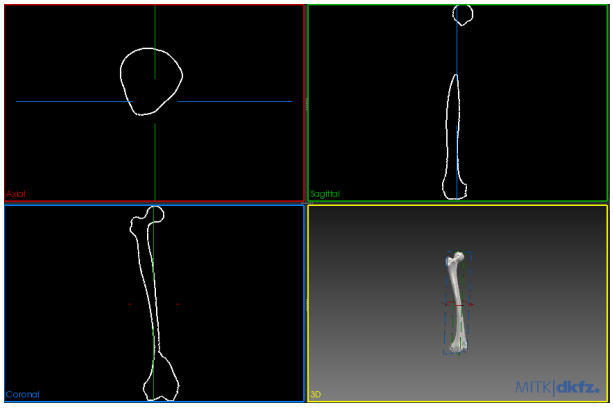


Figura 60: Se pueden observar las vistas 2-D para otra muestra de fémur.

10. Conclusión.

En base a los objetivos presentados al comienzo del presente informe, el trabajo realizado logró satisfacer las necesidades establecidas.

Se logró desarrollar una aplicación, que permite una visualización e interacción adecuada con los objetos posicionados en la escena, no habiendo límite para posicionar el número de objetos deseados en la misma. Se pueden visualizar las coordenadas de cada punto, lográndose un sentido de orientación mediante la adición de ejes cartesianos.

Se cuenta con la capacidad para generar cortes bidimensionales mediante planos axiales, sagitales y coronales sobre los volúmenes, luego de haber configurado MITK para disponer de dicha funcionalidad.

Se pudo lograr un ajuste manual de las piezas mediante un sistema de transformaciones que utiliza rotaciones y traslaciones de forma interactiva, a través del uso del teclado del usuario.

Se pudo cuantificar la cercanía de los objetos alineados mediante una métrica de proximidad global, basada en la distancia euclídea. Dicha métrica está expresada en milímetros debido a las aproximaciones establecidas entre coordenadas globales y milímetros, según explicado en la sección “Error de proximidad”. La métrica permitió reducir, en promedio, unos 2.185 mm la distancia entre los huesos a alinear, representando una mejora en el error de registración global del 37.89 porciento, relativo a una alineación manual previa. Para una correcta apreciación del alineamiento entre piezas se logró, además, implementar un método que permite mostrar un mapa de proximidad entre los objetos alineados. El mismo se basó en los canales espectrales (h,s,v).

En cuanto a la complejidad algorítmica y demoras en la ejecución de los alineamientos automáticos, se logró acelerar el proceso mediante múltiples registraciones en simultáneo y mediante la separación de la interfaz gráfica de las registraciones, haciendo uso de herramientas de programación paralela provistas por el entorno de Qt. Esto permitió operar con la interfaz mientras los demás procesos se ejecutaban de fondo. Se logró visualizar en simultáneo múltiples registraciones (como lo muestra la prueba (0, [2,3,4,5])). Los resultados individuales pueden guardarse como archivos STL, permitiendo que puedan ser cargados en los escenarios virtuales de planeamiento preoperatorio de los especialistas médicos.

Para finalizar, existe la posibilidad de extender este trabajo. Se puede agregar relevancia clínica acoplando este sistema a un banco virtual de huesos. De esta manera, se podría contar con una base de datos que contenga muestras de huesos cadavéricos. Dichos especímenes servirían como potenciales reemplazos. Los métodos desarrollados aquí se pueden acoplar perfectamente a una base de datos, con una pequeña extensión en el desarrollo.

Glosario

- **AR:** Realidad aumentada ("Augmented Reality" en inglés). 8
- **ARAS:** Cirugía asistida por realidad aumentada. 8
- **CAS:** Cirugía asistida por computadora. 4
- **Feature based methods:** Métodos de alineación de imágenes, basados en las características topológicas de las mismas. 11
- **Framework:** Plataforma de desarrollo de software. 11
- **GUI:** Interfaz gráfica del usuario. 18
- **Intensity based methods:** Métodos de alineación de imágenes, basados en la intensidad de los píxeles. 11
- **ITK:** Insight Toolkit, librería de software enfocada en segmentación y registración de imágenes, desarrollada por la compañía Kitware. 11
- **IVN:** Navegación Virtual Intraoperatoria. 5
- **Open Source:** Código de software libre. 11
- **VPP:** Planeamiento Virtual Preoperatorio. 5
- **VTK:** Visualization Toolkit, librería de software desarrollada por la compañía Kitware. 14

Referencias

[1] Adrien Kaladji, Antoine Lucas, Alain Cardon, Pascal Haigron. *Computer-Aided Surgery: Concepts and Applications in Vascular Surgery.* Perspectives in Vascular Surgery and Endovascular Therapy, SAGE, 2012.

[2] Lucas E. Ritacco, Federico E. Milano, Edmund Chao. *Computer-Assisted Musculoskeletal Surgery* Springer, 2015.

[3] Lucas E. Ritacco, Federico E. Milano, Miguel A. Ayerza, DL Muscolo, Fernan González Bernaldo de Quirósa, Luis A. Aponte-Tinaoc. *Bone Tumor Resection: Analysis about 3D Preoperative Planning and Navigation Method Using a Virtual Specimen..* IMIA and IOS Press, 2013.

[4] Lucas E. Ritacco, Federico E. Milano, German L. Farfalli, Miguel A. Ayerza, DL Muscolo, Luis A. Aponte-Tinaoc. *Accuracy of 3-D Planning and Navigation in Bone Tumor Resection.* Orthopedis, 2011.

[5] Luis A. Aponte-Tinao, Lucas E. Ritacco, Miguel A. Ayerza, D. Luis Muscolo, Germán L. Farfalli. *Multiplanar Osteotomies Guided by Navigation in Chondrosarcoma of the Knee* Orthopedics, 2013.

[6] F. Graur, E. Radu, N. Al Hajjar, C. Vaida, D. Pisla. *Surgical Robotics—Past, Present and Future.* International Workshop on Medical and Service, 2018.

[7] Qt Development Frameworks.
Signals Slots. Qt 5.14.0, Qt Core.

[8] Qt Development Frameworks. *Qt Creator Manual.* Qt Documentation.

[9] Kitware.
CMake 3.16 Documentation. CMake Reference Documentation.

[10] Hans J. Johnson, Matthew M. McCormick, Luis Ibanez, and the Insight Software Consortium.
The ITK Software Guide. Fourth Edition, 2019.

[11] Rafael C. Gonzalez, Rachel E. Woods.
Digital Image Processing, Third Edition. Pearson Education International.

[12] Matt McCormick, Stephen Aylward, Julien Jomier and Jerome Velut.
Courses on Medical Image Analysis. Kitware blog, 2015.

[13] Mohammad Rouhani.
Non-rigid Registration between 2D Shapes. Mathworks, 2019.

[14] Jiaolong Yang, Hongdong Li and Yunde Jia.
Solving 3D Registration Efficiently and Globally Optimally. International Conference on Computer Vision (ICCV), 2013

[15] Yun Zeng, Chaohui Wang, Yang Wang, Xianfeng Gu, Dimitris Samaras and Nikos Paragios.
Dense non-rigid surface registration using high-order graph matching. IEEE Conference on Computer Vision and Pattern Recognition.

[16] Chenping Yu, Guilherme C. S. Ruppert, Dan T. D. Nguyen, Alexandre X. Falcao, and Yanxi Liu.
Statistical Asymmetry-based Brain Tumor Segmentation from 3D MR Images. 5th international joint conference on biomedical engineering systems and technologies, 2012

[17] Aether Company. *Aether Announces Launch of AI Software to Advance 3D Organ Printing* MANUFACTURED3D, 2018.

[18] *The VTK User Guide, 11th Edition* Kitware.

[19] P. Vávra, J. Roman, P. Zonča, P. Ihnát, M. Němec, J. Kumar, N. Habib, and A. El-Gendi. *Recent Development of Augmented Reality in Surgery: A Review.* Journal of Healthcare Engineering, 2017.

[20] Sarah Chadwell. *Augmented Reality Simplifies Sinus Surgery Via 3D Mapping* VR Insider, 2017.

[21] Devin Jones. *Digital platform utilizes augmented reality for spinal surgery* Design Engineering, 2019.

[22] Lucas Eduardo Ritaccoa, Alejandro A. Espinoza Oríasc, Luis Aponte-Tinaob, Domingo L. Muscolob, Fernan González Bernaldo de Quirósa, Inoue Nozomu.

Three-Dimensional Morphometric Analysis of the Distal Femur: A Validity Method for Allograft Selection Using a Virtual Bone Bank , 2010 IMIA and SAHIA.

[23] Habib Bou Sleiman, Lucas E. Ritacco, Luis Aponte-Tinao, Domingo L. Muscolo, Lutz-Peter Nolte, Mauricio Reyes.

Allograft Selection for Transepiphyseal Tumor Resection Around the Knee Using Three-Dimensional Surface Registration , Annals of Biomedical Engineering, Vol. 39, No. 6, 2011

[24] Luis Aponte-Tinao, German L. Farfalli, Lucas E. Ritacco, Miguel A. Ayerza, D. Luis Muscolo.

Intercalary Femur Allografts Are an Acceptable Alternative After Tumor Resection , The Association of Bone and Joint Surgeons, 2011.

[25] Lucas Eduardo Ritacco, German Luis Farfalli, Federico Edgardo Milano, Miguel Angel Ayerza, Domingo Luis Muscolo, Luis Aponte-Tinao.

Three-Dimensional Virtual Bone Bank System Workflow for Structural Bone Allograft Selection: A Technical Report , 2013

[26] Ritacco LE, Seiler C, Farfalli GL, Nolte L, Reyes M, Muscolo DL, Tinao LA.

Validity of an automatic measure protocol in distal femur for allograft selection from a three-dimensional virtual bone bank system, 2013

[27] C. E. Ottolenghi.

"mássive osteoarticular bone grafts. Trans- plant of the whole femur", Journal of Bone and Joint Surgery, British, vol. 48, no. 4, pp. 646–659, 1966.

[28] K. S. Conn, M. T. Clarke, and J. P. Hallett

"A simple guide to determine the magnification of radiographs and to improve the accuracy of preoperative templating", Journal of Bone and Joint Surgery, British, vol. 84, no. 2, pp. 269–272, 2002.

[29] D. L. Muscolo, M. A. Ayerza, L. A. Aponte-Tinao, and M. Ranalletta.

"Use of distal femoral osteoarticular allografts in limb salvage surgery. Surgical technique", The Journal of Bone and Joint Surgery, vol. 88, supplement 1, part 2, pp. 305–321, 2006.

[30] Liu Yilin, Xu Huaiyuan, Su Xiu, Liang Haitao, Wang Yi, Chen Xiaodong.

Curvature feature extraction based ICP points cloud registration method, 2018

[31] Natasha Gelfand, Niloy J. Mitra, LeonidasJ. Guibas, HelmutPottmann.

Robust Global Registration, EurographicsSymposiumonGeometryProcessing(2005)

[32] Wolfgang Wittmann*, Thomas Wenger, Benjamin Zaminer, and Tim C. Lueth. *Automatic Correction of Registration Errors in Surgical Navigation Systems, Member, IEEE*