PROYECTO FINAL DE CARRERA
Ingeniería Informática

VNIVERSITAT DE VALÈNCIA

Escola Tècnica Superior d ' Enginyeria
Departament d'Informàtica

Herramienta para modelado estocástico y visualización 3D del sistema de conducción del corazón.

Autor: Víctor Pérez Sainz

Tutor: Rafael Sebastián Aguilar

Septiembre 2011

PROYECTO FINAL DE CARRERA
Ingeniería Informática

VNIVERSITAT DE VALÈNCIA

Escola Tècnica Superior d ' Enginyeria
Departament d'Informàtica

Herramienta para modelado estocástico y visualización 3D del sistema de conducción del corazón.

Autor: Víctor Pérez Sainz

Tutor: Rafael Sebastián Aguilar

Septiembre 2011

Agradecimientos

A mis padres por todo lo que me han dado ya que sin ellos no hubiese sido capaz de llegar hasta aquí.

A mi hermana por todo el apoyo que siempre me ha ofrecido.

A ti Rafa por guiarme con buen pie en esta aventura y dejarme descubrir el mundo de la investigación. Sin tus conocimientos este documento no hubiese sido posible.

A mis amigos y personas queridas por estar ahí siempre.

Resumen

Las enfermedades cardiovasculares fueron en 2008 la principal causa de mortalidad en Europa. El corazón es uno de los órganos más importantes del cuerpo humano y su estudio es fundamental para poder evitar o incluso predecir con anterioridad dichas enfermedades. Sin embargo, la investigación del sistema cardiovascular implica elevados costes tanto económicos como temporales. Por todo ello, el uso de nuevas técnicas computacionales que permitan el estudio y validación de nuevas terapias y fármacos se presenta como una alternativa atractiva desde el punto de vista socio-económico. Nuestro objetivo es contribuir en este aspecto con el desarrollo de un módulo de software que permita el modelado y simulación de la electrofisiológica del corazón y en particular de la inclusión de la red de *Purkinje*, mediante una aplicación desarrollada en C++ e integrada en una plataforma visual de modelado clínica. Con estos modelos se podrán obtener un gran conjunto de datos computacionales que ayuden a entender cuál es el grado de implicación que puedan tener las posibles malformaciones en esta red con dichas enfermedades cardiovasculares, así como optimizar y planificar diferentes terapias que ayuden a paliar sus efectos.

Abstract

The cardiovascular diseases were in 2008 the main cause of mortality in Europe. The heart is one of the most important organs of the human body and its study is fundamental to be able to avoid or to even predict previously these diseases. Nevertheless, the investigation of the cardiovascular system implies high economic costs as much as temporary. By all it, the use of new computation techniques that allow to the study and validation of new therapies and drugs presents/displays like an attractive alternative from the socioeconomic point of view. Our objective is to contribute in this aspect with the development of a module of software that allows the modeled one and simulation of the electrophysiological of the heart and in individual of the inclusion of the network of Purkinje, by means of an application developed in C++ and integrated in a modeled visual platform of clinical. With these models they will be possible to be obtained a great computation data set that helps to understand which is the implication degree that can have the possible malformations in this network with these cardiovascular diseases, as well as to optimize and to plan different therapies that help to palliate their effects.

Contenido

Figuras

Tablas

1- Introducción

El concepto de simulación por computador se desarrolló a la par junto con el rápido crecimiento que tuvieron los computadores y su introducción a gran escala durante los años 40. Gracias a una continua evolución y mejoras, diferentes campos científicos (física, matemáticas, biología…) empezaron a emplear este nuevo potencial para sus estudios. Esto permitió que la informática (ya fuese mediante el diseño de hardware/software específico y más adelante la creación de Internet) entrase en el mundo científico para apoyar y potenciar nuevas líneas de investigación y novedosas aplicaciones. Uno de los campos que más se ha beneficiado de este avance por razones obvias es la medicina.

Gracias al uso del modelado y simulación por computador, podemos estudiar más detalladamente conceptos médicos que anteriormente no podíamos. Este es el caso del modelado de la electrofisiología cardiaca y del sistema de conducción eléctrico del corazón [1]. Éste se encarga de propagar la señal eléctrica generada por el marcapasos natural del corazón, desde la aurícula a los ventrículos, utilizando unas estructuras especializadas con forma de malla. Dicha malla se denomina red de *Purkinje* [2] (fig. 1) y es muy difícil de estudiar, debido a su configuración y sus dimensiones.

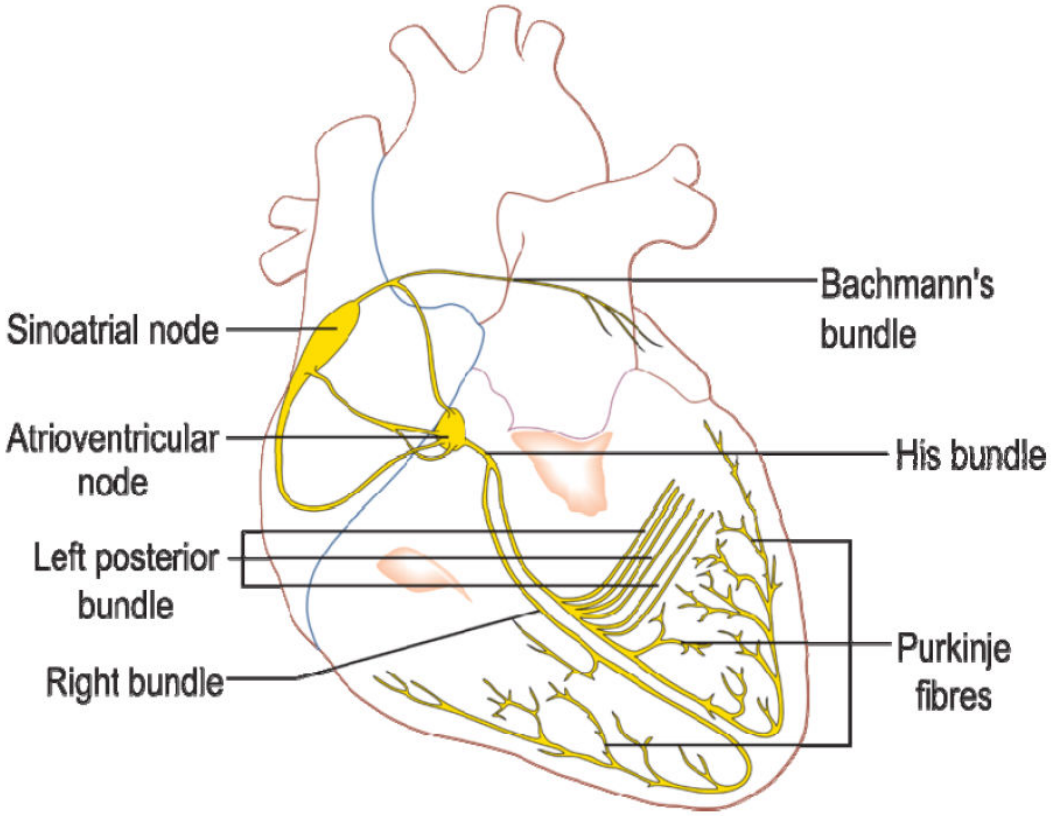

Figura 1. Red de *Purkinje*

Para poder profundizar en el estudio del sistema de *Purkinje*, su estructura geométrica y su función es importante implementar una aplicación gráfica que posteriormente se pueda aplicar para modelado y simulación en cardiología. El objetivo es poder realizar simulaciones 3D de la electrofisiología cardiaca en los que se incluya el sistema de *Purkinje*, y poder variar las propiedades de este modelo, analizando las diferencias por computador. Así mismo, es de vital importancia que las estructuras de *Purkinje* generadas con el modelo sean muy realistas, y permitan estudiar la posible relación que pueda tener esta estructura con las enfermedades cardiovasculares.

Este proyecto forma parte de un proyecto Europeo (euHeart, http://www.euheart.org) de investigación de CommLab junto con CISTIB (Center of Computational Imaging and Simulation Technologies in Biomedicine) en el que participan 17 instituciones de 6 países y que lidera la empresa Philips Research. Está centrado en el uso de técnicas TIC (Tecnologías de la Información y la Comunicación) para mejorar el entendimiento y mejorar el diagnóstico de patologías cardiacas de alto coste e incidencia en Europa.

2- Motivación y Objetivos

Según la OMS [3] (Organización Mundial de la Salud), en 2008 las enfermedades cardiovasculares fueron la primera causa de mortalidad a nivel mundial (fig. 2 y 3).

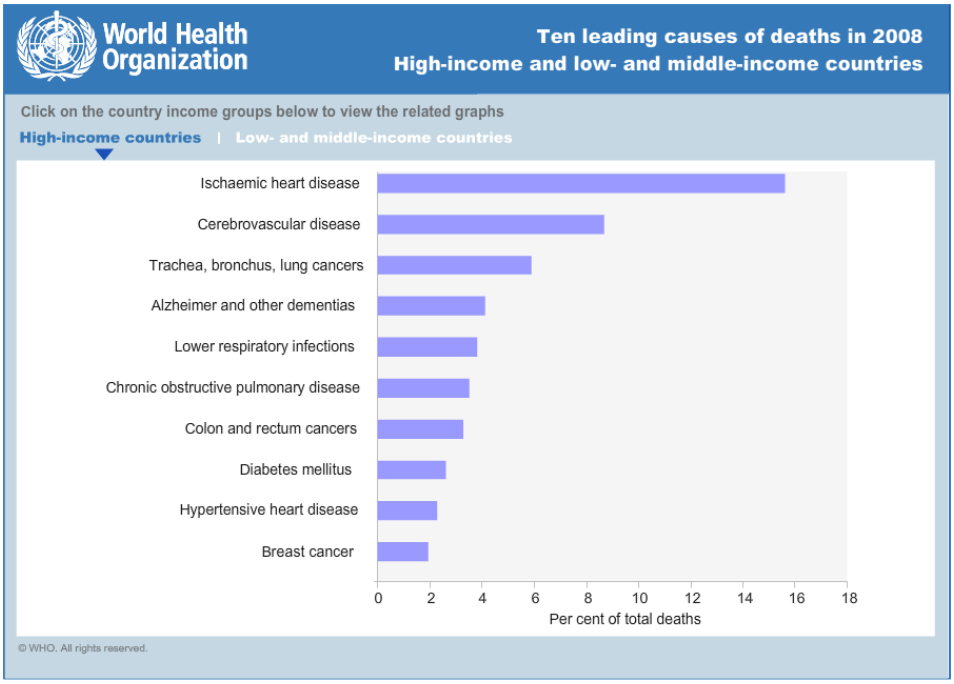

Figura 2. Las 10 causas principales de mortalidad en países desarrollados

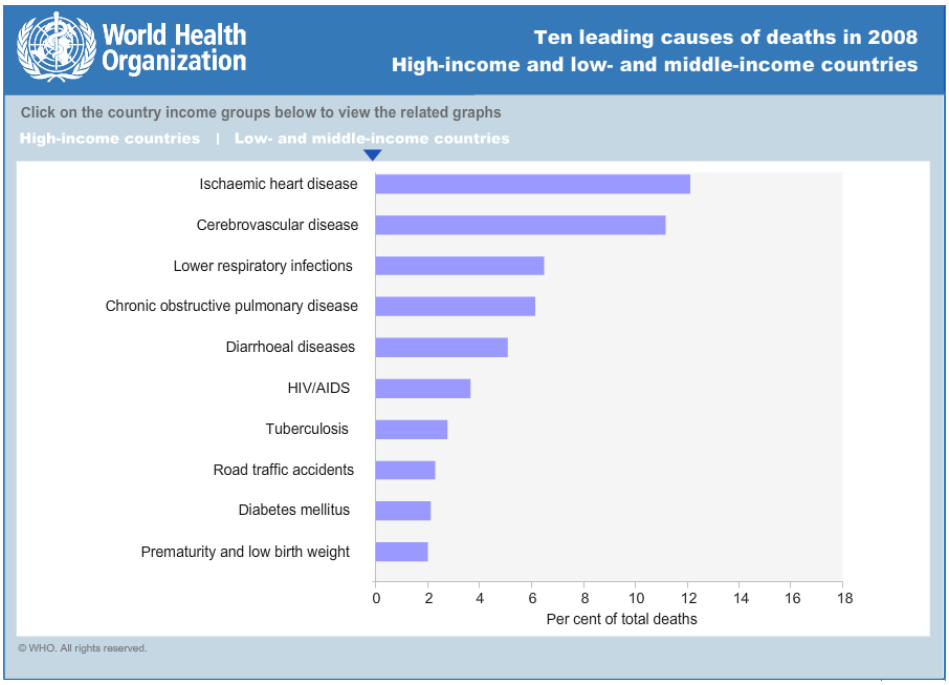

Figura 3. Las 10 causas principales de mortalidad en países subdesarrollados y en vías de desarrollo

Este es un dato a tener muy en cuenta si la tendencia del ser humano es la de alargar la esperanza de vida cada vez más gracias a los avances tecnológicos que se van desarrollando.

Las causas de dichas enfermedades pueden ser diversas como malformaciones, disfunciones o sedentarismo. Hay ciertos aspectos (como el propio sedentarismo) que pueden ser estudiados sin riesgos para los pacientes y tratado para mejorarlo. Pero hay otras causas que son más difíciles de estudiar y que requieren mucho más tiempo y medios. Por esa razón no disponemos de muchos detalles sobre la microestructura del corazón.

Estos estudios son caros y conllevan un elevado grado de complejidad. Se necesita una gran inversión de tiempo y recursos económicos para conseguir obtener resultados que supongan avances importantes. Haría falta un gran banco de pruebas, pero por razones legales y éticas no sería posible investigar ciertos aspectos en individuos vivos. Es por eso que vamos a crear una aplicación gráfica que ayude en la realización de estudios por computador.

Más concretamente nos vamos a centrar en la estructura que hemos citado anteriormente: la red de *Purkinje*. Esta red, que está presente en el corazón de los mamíferos, se podría entender como un cableado eléctrico que transporta pulsos eléctricos a gran velocidad por las paredes musculares del corazón y que sirven para sincronizar la contracción, con el fin de bombear de forma eficiente la sangre al resto del cuerpo. No se conocen todos los aspectos morfológicos y funcionales de esta estructura, pero se conoce su rol en determinadas patologías. Para poder experimentar e intentar vislumbrar la influencia de este sistema, desarrollaremos algoritmos que permitan la construcción de estructuras realistas 3D del sistema de *Purkinje*, que incluiremos en modelos humanos del corazón reconstruidos previamente a partir de técnicas de imagen como resonancia magnética (RMI) o tomografía axial computerizada (TAC). Finalmente estos modelos 3D se utilizarán para realizar simulaciones 3D de la electrofisiología cardiaca.

No se conocen muchos detalles de esta estructura o de su funcionamiento, puesto que no se ha podido estudiar su funcionamiento *in-vivo* debido a limitaciones técnicas. La poca información que se tiene, se ha obtenido mediante imágenes de cortes de secciones de corazón (fig. 4) en las que se aplicó un tinte para un mejor visionado e identificación.

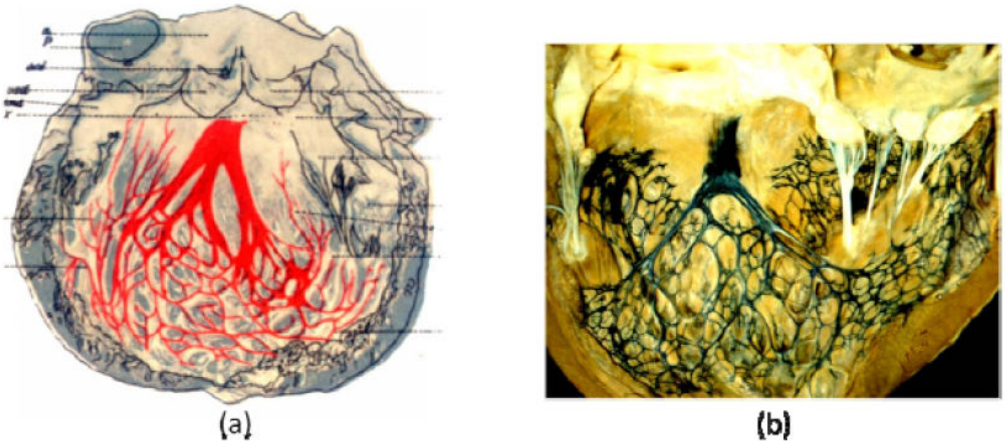

Figura 4. Red de *Purkinje* del corazón. a) ilustración de Tawara mostrando las tres divisiones en el LBB humano. b) red de *Purkinje* tintada en ternero, proporcionada por el Prof. Sanchez-Quintana

Gracias a estas imágenes tenemos una ligera idea de su estructura física pero no podemos asegurar como pueden repercutir en su funcionamiento los posibles problemas o fallos en la red. Al igual que la mayoría de grupos de investigación con presencia en el ámbito del modelado eléctrico del corazón, no tenemos actualmente un modelo de *Purkinje* preciso o suficientemente realista. Por ello con este proyecto se pretende desarrollar un modelo paramétrico del sistema de *Purkinje*, así como un entorno interactivo en el que se pueda desplegar el sistema de *Purkinje* en un modelo 3D de un ventrículo.

La simulación por computador [4] a veces complementa o incluso sustituye a los estudios empíricos o experimentales para los que no es posible hallar soluciones analíticas. Existen diferentes tipos de simulación pero el propósito general de todas ellas es el tratar de generar un conjunto de escenarios representativos utilizando un modelo puesto que una relación completa de todos los estados posibles de éste sería muy costosa o imposible.

Gracias a estas simulaciones, podemos tener un gran banco de pruebas sin tener ningún problema tanto legal como ético. Mediante la ejecución de la aplicación con diferentes parámetros de entrada, obtendremos un banco de pruebas muy variado del que poder sacar las estadísticas necesarias para obtener resultados aceptables. Con esto vamos a poder estudiar mejor la relación existente entre la red de *Purkinje* y las posibles enfermedades derivadas.

Para que esto sea posible y sus resultados válidos nuestro modelo de *Purkinje* tiene que ser lo más real posible. Es decir, hay que transferir en la medida de lo posible los conocimientos biológicos e histológicos disponibles hasta la fecha al modelo. Hay que tener en cuenta tanto lo más obvio, como la geometría de los ventrículos o las propiedades del tejido, como los detalles más finos, relacionados a estructuras microscópicas o no observables con técnicas de imagen clínica. De esta manera será posible trasladar los resultados obtenidos en el ámbito computacional al ámbito clínico.

Para construir una estructura anatómicamente realista se pueden emplear modelos y algoritmos informáticos ya publicados en revistas de investigación. Aun así siempre existirán ciertas limitaciones puesto que hay detalles del corazón que no se conocen completamente. El modelo que se propone para abordar el problema definido, consiste en un algoritmo basado en lenguajes naturales que ha sido propuesto en otros ámbitos para modelar estructuras arbóreas, conocido como *L-system* [5] [6]. Estos árboles comienzan en un punto determinado (sabemos donde se localiza), para crear tres sucesores o nodos hijos que a su vez tendrán múltiples sucesores y crearán ramificaciones de la estructura inicial. Esta estructura va creciendo produciendo nuevas generaciones (sucesores) que han de seguir un conjunto de reglas codificadas en el modelo y que aseguran el realismo de la estructura final. Dichos conjuntos de reglas vienen determinadas por el usuario a través de parámetros de entrada, que por tanto podrá influir en la morfología del sistema de *Purkinje* que se genere.

El objetivo principal del proyecto es la creación de una aplicación junto con un *plugin* gráfico que permita usarla de forma sencilla y clara. La idea es que no se necesite una gran cantidad de tiempo de aprendizaje para saber usarla y de esta manera pueda ser utilizada por usuarios que no tengan un elevado grado de conocimiento en informática. Vamos a explicar un poco más en detalle los puntos más importantes.

- **Investigación de la estructura de *Purkinje*:**

Gracias a este análisis podremos ser capaces de aprender cómo es la estructura real de *Purkinje* y trasladar estos conocimientos a un algoritmo mediante un conjunto de reglas.

- **Conocimientos de modelos de *Purkinje* actuales:**

Actualmente sabemos desde donde nace, como se expande y aproximadamente donde se localiza en el corazón. Desconocemos cómo crece exactamente y que distribución sigue en este proceso. La mayoría de modelos existentes se basan en delineaciones manuales muy simplificadas del sistema de *Purkinje* que por lo general requieren unos conocimientos avanzados del mismo. Además su delineación requiere una gran inversión de tiempo por parte del usuario, que por lo general no puede volver a reutilizar la estructura para otros modelos geométricos del ventrículo.

- **Nuevo modelo/algoritmo basado en *L-systems*:**

L-systems es un algoritmo recursivo de reescritura en paralelo empleado para generar modelos de crecimiento de plantas o de modelos orgánicos. En nuestro caso nos servirá para hacer “crecer” nuestra red de *Purkinje* alrededor del corazón. Además nos permitirá incluir una serie de reglas que asegure un resultado más fisiológico, evitando que el usuario tenga que conocer en detalle la estructura real del sistema.

- **Parametrización del algoritmo para resultados realistas:**

Lo que queremos es que este algoritmo resulte lo más realista posible dentro de los conocimientos que tenemos sobre como es la red de *Purkinje*. Es por ello que disponemos de un elevado número de parámetros de entrada para intentar controlar completamente como se va a generar la estructura, su densidad, el número de ramas, etc. El usuario podrá generar distintos sistemas en función de estos parámetros y valorar posteriormente el realismo del mismo mediante simulaciones eléctricas.

- **Plug-in gráfico (dentro de un entorno de prototipado médico GIMIAS):**

El módulo de construcción del sistema de *Purkinje* estará controlado por un *plugin* o interfaz gráfica. Dicha interfaz nos va a permitir ver el resultado de nuestro algoritmo para poder estudiar el resultado. Dispondremos de un control de tiempo que nos permitirá ver cómo va evolucionando la estructura así como una serie de menús que se detallarán más adelante. Todo esto dentro de un entorno llamado GIMIAS [7].

- **Inclusión del modelo para simulaciones electrofisiológicas cardiacas:**

La objetivo final es que las estructuras que se generen puedan ser directamente incluidas en modelos de electrofisiología que utilicen entornos de simulación paralelo basado en elementos finitos u potencien su aplicación en el estudio y la investigación de enfermedades cardiovasculares. Como ejemplo se estudiará el caso de ver como se comportaría nuestra red de *Purkinje* en un modelo de ventrículo izquierdo utilizando diferente densidad de ramas.

3- Estado del arte

La simulación por computador ha ido evolucionando a lo largo de los años para ir ofreciendo información más detallada, precisa y fiable. Esto ligado a la constante evolución [8] que ha ido teniendo la medicina desde 1900, provocó que en los años 70-80 tuviésemos las primeras simulaciones médicas gracias a la aparición, en el mercado, de los microprocesadores.

Al principio, los primeros computadores [9] generaban imágenes en 2D y realizaban cálculos sencillos puesto que los ordenadores de esa época tenía una limitación alta en cuanto a potencia de cálculo. Si queríamos más potencia necesitábamos acceder a las instalaciones que disponían de varios computadores dispuestos en paralelo.

Sobre 1995 aparecieron las primeras tarjetas gráficas 2D/3D. Con esto se consiguió separar los cálculos necesarios para generar espacios tridimensionales en tiempo real de la CPU a la GPU. Ahora se disponía de un microprocesador único para realizar sólo los cálculos matemáticos y procesar toda la información.

Esto propició un gran avance en las simulaciones al permitir realizar modelos 3D rápidos y con los que poder interactuar (rotar, zoom, modificar, etc…). Tanto las GPUs como las CPUs continúan evolucionando a un ritmo vertiginoso. Con respecto a las CPUs, han ido disminuyendo de tamaño mientras aumentaban las prestaciones. Ahora tenemos procesadores multicore lo que nos da la opción de tener un mini centro de proceso en un espacio de unos pocos milímetros. En cuanto a las GPUs, se han ido aumentado su potencia de cálculo y brindan la posibilidad de conectar tarjetas gráficas en paralelo.

Estos avances han tenido mucha repercusión en diversas áreas de investigación. Tanto en física aplicada como en química o biología o el área de la medicina.

Los modelados computacionales aplicados a la medicina han tenido un boom en los últimos años. Anteriormente como ya hemos comentado eran modelos simples y en 2D. Ha sido estos últimos 10-15 años cuando han empezado a tener más relevancia gracias a la aparición de los modelados tridimensionales y la posibilidad de llevar a cabo simulaciones complejas. Con las simulaciones 3D podemos representar casi cualquier cosa que se nos ocurra para poder realizar estudios más detallados y que de otra manera serían inviables (fig. 5).

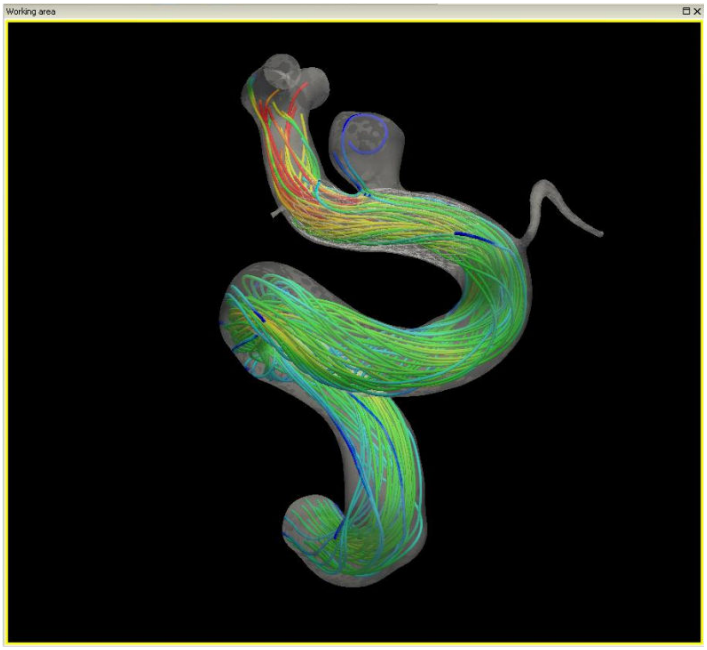

Figura 5. Simulación de fluidos en modelo de vaso sanguíneo

Algunas nuevas tendencias a nivel internacional como Physiome (http://www.physiome.org/) o a nivel europeo como el Virtual Physiological Human (VPH, http://www.vph-noe.eu/) han apostado por el uso de modelos ICT para el modelado del organismo humano de manera virtual. Existen simulaciones de múltiples órganos, y para cada uno de ellos con diferentes físicas. Así nosotros estamos interesados en el modelado del corazón y en particular del comportamiento de su física eléctrica. Otras físicas del corazón relevantes son el estudio de su mecánica o del estudio de fluidos. La mayoría de los modelos suelen estar orientados a ciertos aspectos del corazón, no a su globalidad, y se habla del modelado multiescala, es decir, se modela cada física a distintos niveles de resolución acoplados como son, célula, tejido y órgano.

Es por ello que durante los últimos años se ha promovido la introducción de técnicas de modelado por computador tanto en el área de imagen médica como en el área de simulación biofísica. Modelos más precisos y nuevas líneas de financiación en el programa marco europeo han facilitado la expansión e introducción de estas técnicas de modelado, restringidas a áreas de investigación, en empresas y entornos clínicos.

En este capítulo, explicaremos los modelos computacionales de *Purkinje* más importantes utilizados actualmente. Posteriormente, veremos de qué herramientas disponemos (entornos gráficos) para generar la red de *Purkinje*. Para finalizar revisaremos los software de prototipado médico más conocidos en este ámbito.

3.1- Modelado del corazón por computador

El corazón (fig. 6) es uno de los órganos más complejos que existen. Ya no sólo por su importancia sino también por las estructuras que lo componen y su funcionamiento.

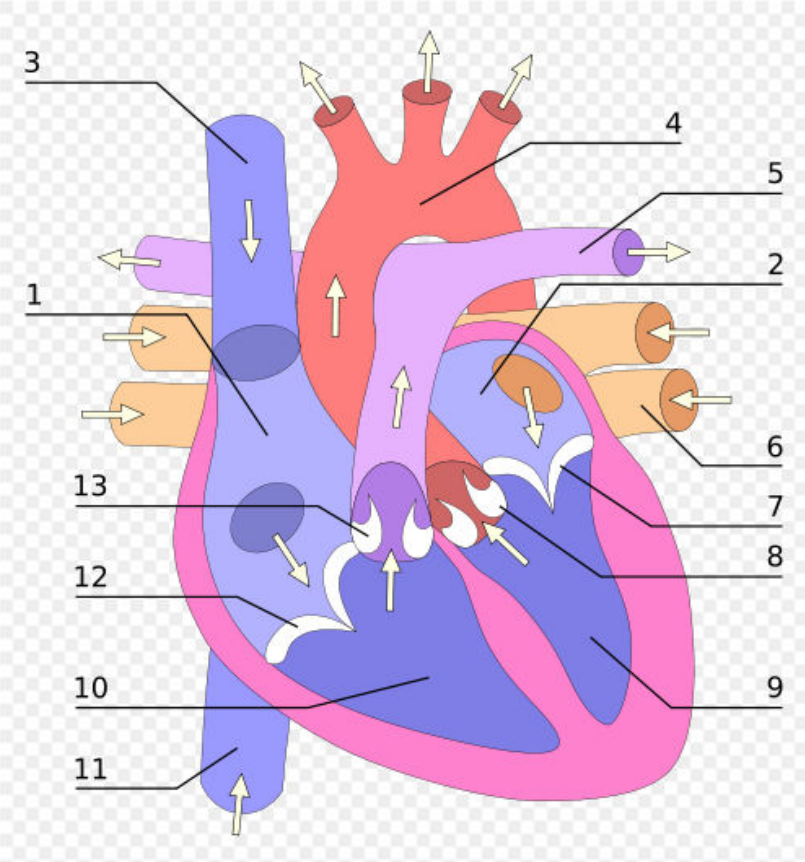

Figura 6. 1. Atrio derecho, 2. Atrio izquierdo, 3. Vena cava superior, 4. Aorta, 5. Arteria pulmonar, 6. Vena pulmonar, 7. Válvula mitral, 8. Válvula aórtica, 9. Ventrículo izquierdo, 10. Ventrículo derecho, 11. Vena cava inferior, 12. Válvula tricúspide, 13. Válvula pulmonar

Como podemos ver en la figura superior, el corazón se compone de una gran cantidad de partes y estructuras (sólo muestra las estructuras generales sin detallar). Las paredes ventriculares están formadas por tres regiones básicas: endocardio (una membrana serosa de endotelio y tejido conectivo de revestimiento interno, con la cual entra en contacto la sangre. Incluye fibras elásticas y de colágeno, vasos sanguíneos y fibras musculares especializadas, las cuales se denominan Fibras de *Purkinje*. En su estructura encontramos las trabéculas carnosas, que dan resistencia para aumentar la contracción del corazón), miocardio (es una masa muscular contráctil. El músculo cardíaco propiamente dicho; encargado de impulsar la sangre por el cuerpo mediante su contracción. Encontramos también en esta capa tejido conectivo, capilares sanguíneos, capilares linfáticos y fibras nerviosas) y epicardio (es una capa fina serosa mesotelial que envuelve al corazón llevando consigo capilares y fibras nerviosas. Esta capa se considera parte del pericardio seroso).

Es muy importante, para aumentar la esperanza de vida del ser humano, conocer con el máximo detalle posible el funcionamiento de este órgano y

estudiar las causas de las enfermedades cardiovasculares y como pueden influir las malformaciones provocadas sobre estas. Este conocimiento es muy difícil de conseguir tanto por la complejidad del propio órgano, como por razones técnicas y éticas. Es por ello que en muchos casos se utiliza información extraída de experimentos sobre órganos "muertos" o *ex vivo*. De estos estudios se obtiene principalmente información histológica, es decir, sobre las estructuras, pero el comportamiento a nivel funcional en condiciones patológicas es todavía difícil de comprender.

En relación a las fibras de *Purkinje*, su existencia y función (conducir el potencial eléctrico generado en las aurículas hasta los ventrículos y activar el tejido ventricular) se descubrieron a principios del siglo pasado, pero su estudio en detalle y su modelado son todavía un problema no resuelto.

La aparición de los primeros modelos biofísicos del corazón junto con el uso de simuladores permitió empezar a realizar test más complejos sin necesidad de llevar a cabo experimentos *in-vivo*, aunque en contrapartida requiriesen un elevado coste computacional. Esto provocó un gran avance puesto que ya no era del todo necesario el uso de sujetos vivos para la investigación. Aun así es importante remarcar las limitaciones inherentes de los modelos en los que no se pueden testear todo las hipótesis ni modelar todos los escenarios existentes.

El uso de computación paralela y de tarjetas gráficas está dando lugar a una gran revolución en cuanto a las simulaciones médicas. Ahora además de hacer pruebas, se pueden hacer simulaciones en modelos 3D para ver con más claridad su hipotético comportamiento. Esto va a permitir a los investigadores en un horizonte no muy lejano la visualización global del corazón, y el estudio de su función sobre un modelo para una gran cantidad de escenarios. Gracias a esto se obtenían grandes bancos de pruebas de los experimentos minimizando enormemente el coste de estos.

La tecnología actual de multicore y GPUs avanzadas permiten hacer simulaciones 3D muy realistas a nivel estructural, que requieren miles de millones de cálculos para generarlas, en tiempos aceptables (horas o días) (fig. 7).

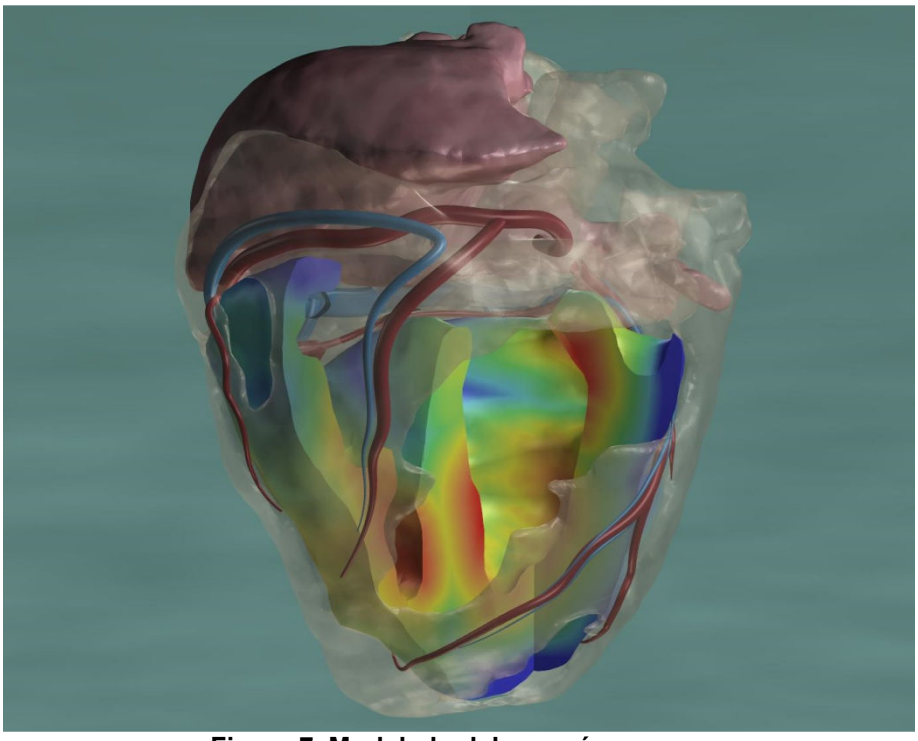

Figura 7. Modelado del corazón

En los últimos 10 años se están llevando a cabo experimentos sobre múltiples conceptos para estudiar los resultados generados y conocer un poco mejor este importantísimo órgano. Por ejemplo tenemos simulaciones sobre el latido del corazón (referencia), como afecta la temperatura al miocardio (referencia), la actividad eléctrica (referencia), flujo del movimiento de la sangre (referencia), hasta corazones virtuales que engloban todo.

Aunque va más allá del ámbito del modelado del corazón, es importante remarcar que estos grandes avances también han permitido mejoras a nivel de la cirugía cardiovascular. Gracias a estas simulaciones, las operaciones actuales se pueden realizar con el mínimo riesgo posible para el paciente (teniendo en cuenta el alto nivel de dificultad de estas operaciones) puesto que se pueden ensayar en las simulaciones hasta conseguir los resultados deseados. Se entiende que en la realidad pueden producirse dificultades imprevistas durante la intervención pero al menos se parte de una idea más clara y completa de lo que se tenía anteriormente.

3.2- Modelos computacionales de Purkinje

La estructura de la red de *Purkinje* es bastante compleja y no se tienen muchos detalles de cómo se genera. Se sabe que existe gracias a estudios histológicos que se realizan en corazones extraídos y que posteriormente se tintan para su visualización. Hasta la fecha no se tienen imágenes detalladas de *Purkinje* en corazones vivos en el ser humano. Es por ello que los modelos que se han desarrollado son generalmente aproximaciones y no se pueden considerar realistas.

Los primeros modelos se basaban en modelos de estimulación del tejido en el que se distribuían pequeños electrodos o estímulos en el endocardio. Posteriormente se comenzó la generación de árboles partiendo de un nodo principal y desarrollando siempre la estructura con el mismo ángulo y orientación. Estos primeros modelos resultaron ser una pequeña aproximación, pero había que tener ciertos datos en cuenta, vistas las imágenes y constatar que no siempre tienen la misma distribución.

Posteriormente se comenzó a programar la generación de esta red mediante un sistema denominado *L-system*. Más adelante explicaremos en que cosiste dicho método y veremos algunos ejemplos de crecimiento de las ramas según ciertas características.

3.2.1- Primeros modelos

Los primeros modelos eran relativamente simples. Solo tenían en cuenta ciertas características básicas para generar la red de *Purkinje*, lo cual se consideró algo normal puesto que el sistema de conducción cardiaco es muy complejo. Hay que modelar muchas peculiaridades de las estructuras por lo que se necesitan ciertos conocimientos de la misma a nivel histológico y cierto potencial de cálculo para su modelado por computador.

Los primeros sistemas que simulaban la conducción eléctrica por el corazón carecían de cualquier estructura y sólo estaban orientados a simular los patrones de activación eléctrica descritos en otros estudios electrofisiológicos por medios de puntos de activación. Estas técnicas aún se usan en algunos casos para generar una hipotética red de *Purkinje* densa e infinita.

En si las primeras simulaciones que incluyen ciertas estructuras datan de los años 80. Generadas en imágenes 2D, ya permitían crear las ramas que iban creciendo según una distribución en árbol. Aoki [10] desarrollo un modelo que se componía de aproximadamente 50000 células organizadas dentro de una estructura circular cerrada según los datos anatómicos que se tenían. El sistema de conducción estaba compuesto de las ramas principales y sus hijos que constituyen las fibras de *Purkinje*.

Abboud [11] desarrolló un modelo 2D de elementos finitos de los ventrículos con un sistema de conducción fractal. Los dos ventrículos se modelaban como esferas formadas por aproximadamente medio millón de elementos. La red de *Purkinje* se modeló como un árbol asumiendo que la longitud de cada rama del mismo tipo y el ángulo con el que se generan es siempre el mismo para cada generación. El modelo fue creado para simular el complejo QRS de alta definición del ECG bajo condiciones normales e isquemia (fig 8).

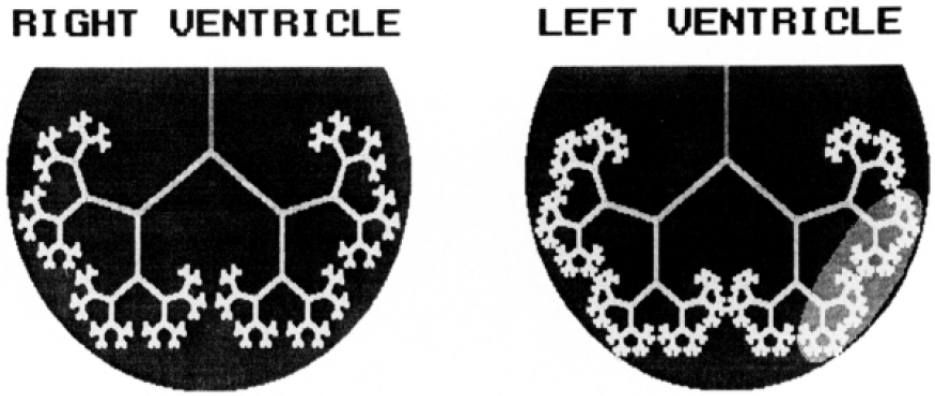

Figura 8. Representación del modelo con una región dañada representada por un coloreado gris claro

Pollard y Barr [12] desarrollaron un modelo anatómico usando 35000 elementos cilíndricos aunque no estaba incorporado en un modelo ventricular. El modelo se validó con datos con tiempo de activación. Este modelo utilizaba 35 puntos de activación en el endocardio de los cuales sólo 4 estaban extraídos de estudios de activación cardiaca y se dibujaron una serie de cables para conectarlos entre sí. Dichos cables estaban altamente refinados para crear pequeños elementos que permitiese la simulación de la conducción eléctrica. Los cables estaban conectados a otros cables para formar bucles.

Berenfeld y Jalife [13] propusieron un modelo que incluía 214 puntos de unión entre las fibras de *Purkinje* y el músculo del corazón. Para imitar la geometría de las ramas principales de *Purkinje* utilizando tintes para su visualización (fig. 9).

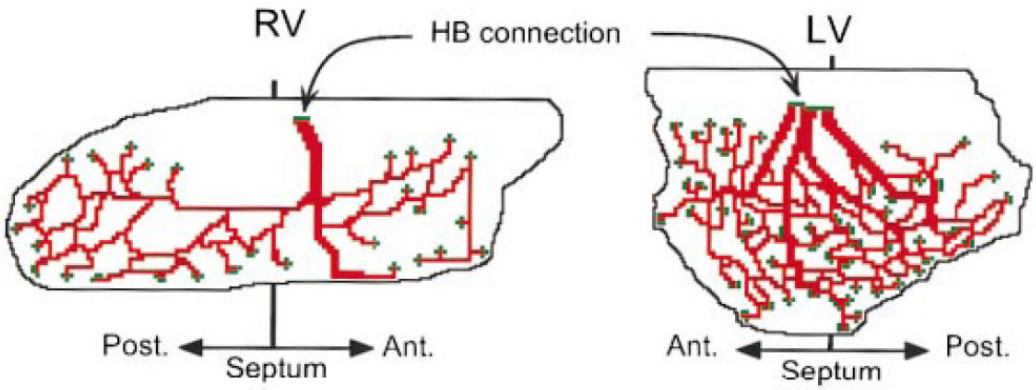

Figura 9. Representación de la parte derecha e izquierda de la red de *Purkinje* superpuestas a la superficie de la superficie endocardíaca

Más recientemente, Simelius [14] delineó un modelo de *Purkinje* para el ventrículo humano minimizando las diferencias entre las isócronas simuladas y las medidas hechas por Durrer (fig. 10a). De forma similar a esta simulación, Ten-Tusscher y Panfilov crearon un sistema de conducción periférico usando un sistema similar que incluida 214 puntos de estimulación y seguían descripciones disponibles en la literatura para emplazar los bloques principales (fig. 10b).

Vigmond y Clements [15] desarrollaron un modelo de *Purkinje* dibujando de forma manual (en 2D) la estructura en árbol y posteriormente superponiéndola al correspondiente ventrículo en 3D.

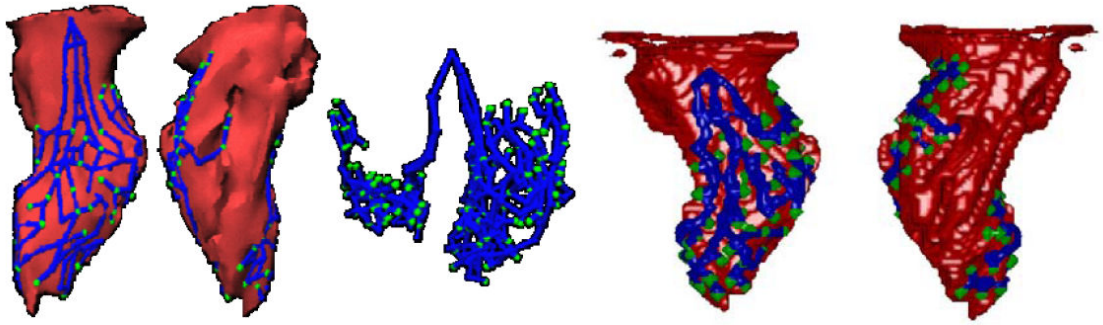

Figura 10. Geometría de la red de *Purkinje*. a) Modelo de Simelius. b) Modelo de Ten-Tusscher. Los puntos verdes corresponden a l unión red-músculo

Como podemos ver se han creado bastantes modelos informáticos para generar la red de *Purkinje*. Aunque todos ellos emulaban la conducción eléctrica por el corazón, era necesario desarrollar un modelo con una estructura más realista y que permitiese de forma dinámica y automatizada crear redes complejas. Es por esto que se empezó a emplear *L-system* para que el crecimiento fuese más parecido a la realidad.

3.2.2- Modelos basados en *L-system*

L-system es una gramática formal originalmente presentada por Lindenmayer para formalizar el desarrollo de las plantas multicelulares [16] y se amplió posteriormente para representar plantas superiores y estructuras de ramificaciones más complejas (fig. 11). El *framework* del *L-system* consiste en una estructura inicial y unas reglas de reescritura (o la generación de reglas). La esencia del desarrollo consiste en el reemplazo en paralelo utilizando las reglas de reescritura. A partir de la estructura inicial, el *L-system* reemplaza cada parte de la estructura actual mediante la aplicación de la regla de forma secuencial.

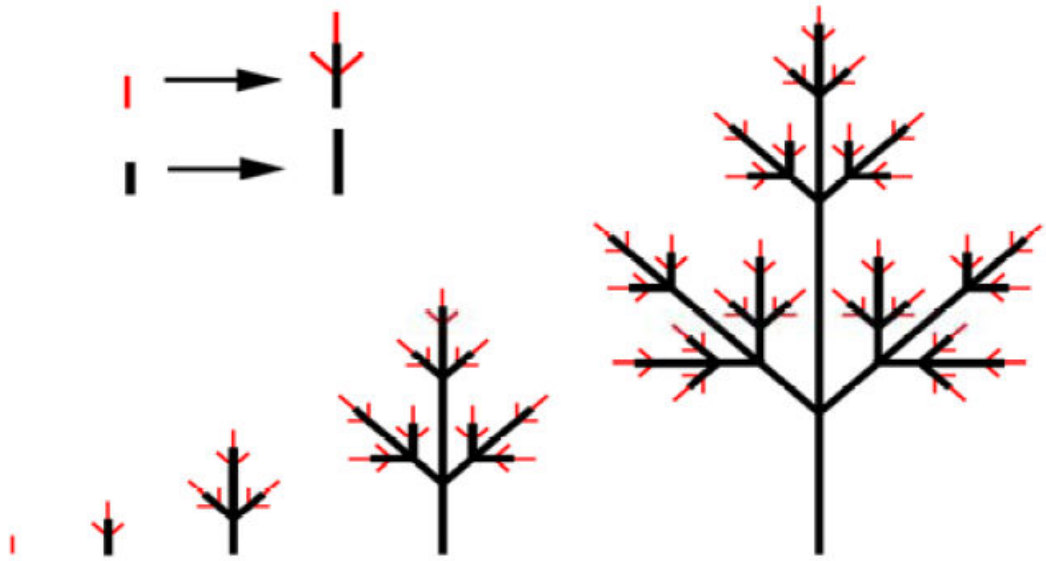

Figura 11. Ejemplo simple del *L-system* (crecimiento de una hoja). Partiendo de una estructura simple, va generando la estructura aplicando dos reglas de crecimiento (arriba izq) secuencialmente

La figura superior muestra un ejemplo simple con el desarrollo de una hoja común. Esto incluye dos tipos de módulos: los vértices (líneas rojas) y los internodos (líneas negras). El ejemplo tiene dos reglas simples (esquina superior izquierda): la primera reemplaza un vértice por un internodo con tres vértice más (izquierda, derecha y cetro) y la segunda sustituye un internodo por dos juntos (uno encima del otro). La estructura inicial es un vértice simple. Mediante la aplicación de las dos reglas descritas, el sistema desarrolla una intrincada estructura de rama.

Un aspecto interesante del sistema, es que cada proceso de reemplazo corresponde al crecimiento de una parte de la planta. Por lo tanto, el *L-system* no es solamente una técnica heurística que crea formas de tipo fractal, es también una simulación de crecimiento de las plantas en el mundo real. Karch presento un método similar para generar patrones de vasos de árbol. Sin embargo el *L-system* y el método de Karch están diseñados para estructuras abiertas de árboles y no se pueden aplicar de forma directa a las estructuras de malla cerradas vistas en las fibras de *Purkinje*.

Hay que extender el algoritmo añadiendo reglas para posibles bucles (que no existen para los árboles) entre las ramas así como añadir las variaciones de los ángulos de crecimiento.

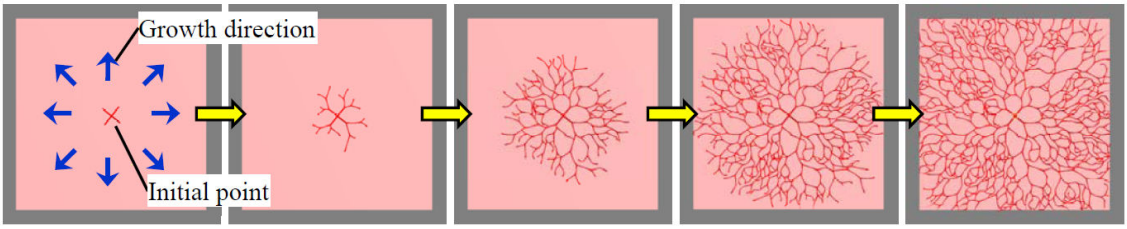

Figura 12. Crecimiento de las fibras de *Purkinje* según *L-system*

En la anterior figura (fig. 12) podemos ver cómo sería el proceso de crecimiento de las fibras de *Purkinje* de nuestro modelo. Como podemos ver el inicio puede ser relativamente lento pero conforme vamos creciendo, el número de ramas crece de forma exponencial y puede acabar rápidamente con el espacio existente. Cabe hacer notar que las ramas son totalmente dinámicas, es decir, no siguen todas las mismas distribuciones, ni parámetros para generarse. Suelen ser diferentes. Esto lo conseguimos gracias a diferentes reglas de reemplazo.

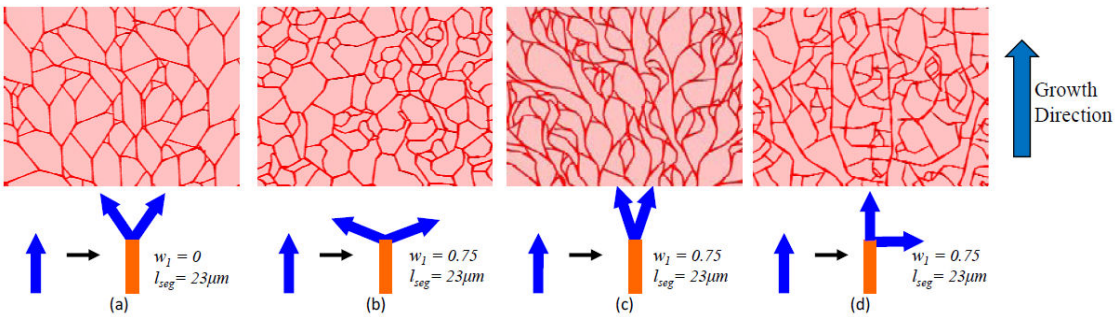

Figura 13. Diferentes reglas de crecimiento

En esta figura (fig. 13) podemos una serie de reglas de ejemplo y el resultado que obtendríamos al aplicarlas. Como vemos al variar el ángulo de crecimiento y teniendo la misma longitud, obtenemos conjuntos diferentes.

El *L-system* puede ser un gran avance en lo que respecta la generación de la estructura de *Purkinje*. Aunque desconozcamos si de verdad se genera de esta manera o no, es una aproximación bastante viable para realizar simulaciones y comprobar la influencia que pudiese tener según los parámetros de crecimiento que le hayamos introducido. Ya está siendo empleado por varias aplicaciones

Uno de los investigadores que actualmente está desarrollando software con esta idea es el Dr Takashi Ijiri de la universidad de Tokio. Está generando un modelo del corazón en 3D.

3.3- Plataformas de modelado y simulación médica

Con la entrada de los gráficos 3D se han ido creando un conjunto de herramientas de modelado y simulación médica que se han ido englobando en una serie de plataformas más grandes. La idea es tener en una sola aplicación, todas las herramientas para el modelado de diferentes áreas del cuerpo humano.

Nuestra aplicación se ha diseñado para ser integrada en una plataforma de prototipado médico denominada GIMIAS, que explicaremos más adelante. Por ahora vamos a centrarnos en las que existen actualmente.

3.3.1- CardioSolv

Software desarrollado para la simulación del corazón. Tiene dos productos: “Cardiac Arrhythmia Research Package (CARP)” [17] y “Tarantula”.

CARP consta de una librería validada de modelos iónico celulares (LIMPET) con una herramienta de simulación de una única célula (bench) y un simulador de tejidos. En particular los modelos iónicos de, Luo-Rudy, Shiferaw-Mahajan y Ten-Tusscher I y II han sido ampliamente utilizados y validados. La herramienta de simulación de una sola célula es un programa de comandos de línea de Linux/Unix y puede ser usado para generar varios tipos de simulación. Todos los modelos de la librería LIMPET tienen que ser cargados en el simulador CARP para una completa simulación 3D de los tejidos.

El simulador CARP se puede ejecutar en cualquier máquina con al menos 1 y hasta 16000 CPUs, gracias al uso de la biblioteca de paralelización MPI. Puede manejar desde modelos de una célula hasta millones de ellas y se ha usado para simular el corazón humano. LIMPET, se ha diseñada para ejecutarse en sistemas Linux/Unix y escala muy bien en interconexiones de baja latencia como Infiniband y Myrinet. CARP es utilizado actualmente por varios de los laboratorios académicos líderes en el campo de la simulación cardiaca y también se ha usado para simulaciones cardiacas comerciales.

CardioSolv apoya la herramienta de mallado “Tarantula”, desarrollada por CAE Software Solutions. Es esencial para la generación de mallas de elementos finitos que representen de forma detallada los tejidos del corazón y puede usarse conjuntamente con CARP.

3.3.2- Chaste

Chaste (Cancer, Heart and Soft Tissue Environment) [18] es un paquete de simulación de propósito general destinado a problemas multiescala y computacionalmente exigentes que surgen en biología y fisiología. La funcionalidad actual incluye simulación de tejidos y electrofisiología a nivel celular, modelado de tejidos discretos y blandos. El paquete está siendo desarrollado por un equipo basado principalmente en el grupo de biología computacional en el laboratorio de computación de la universidad de Oxford. Se basa en la experiencia de la ingeniería de software, computación de alto rendimiento, los modelos matemáticos y la informática científica.

Mientras que Chaste es una librería genérica extensible, el desarrollo de software hasta la fecha se ha centrado en dos áreas distintas: modelado continuado de la electrofisiología cardiaca (Cardiac Chaste) y el modelado discreto de poblaciones de células (System Biology Chaste) con aplicación específica a la homeostasis del tejido y la carcinogénesis (Cancer Chaste).

3.3.3- Mimics Innovation Suite

Conjunto de aplicaciones compuesto por tres herramientas principales que pueden ser usadas de forma independiente, aunque dan un mejor resultado cuando se usan de forma conjunta.

Estas tres herramientas son: Mimics, 3-matic y un conjunto de servicios para ingeniería, consultoría y desarrollo de software personalizado [19].

Mimics (fig. 14) permite procesar los datos de imágenes en 2D (CT, μCT, MRI, etc…) para construir modelos en 3D con la máxima precisión, flexibilidad y facilidad de uso. Las herramientas de segmentación de gran alcance permiten segmentar imágenes médicas CT/MRI, tomar medidas y acciones de ingeniería directamente en el modelo 3D. Desde la aplicación se puede exportar dicho modelo 3D bajo una amplia gama de formatos de salida y aplicaciones de ingeniería tales como FEA, diseño, simulación quirúrgica, fabricación de aditivos y más.

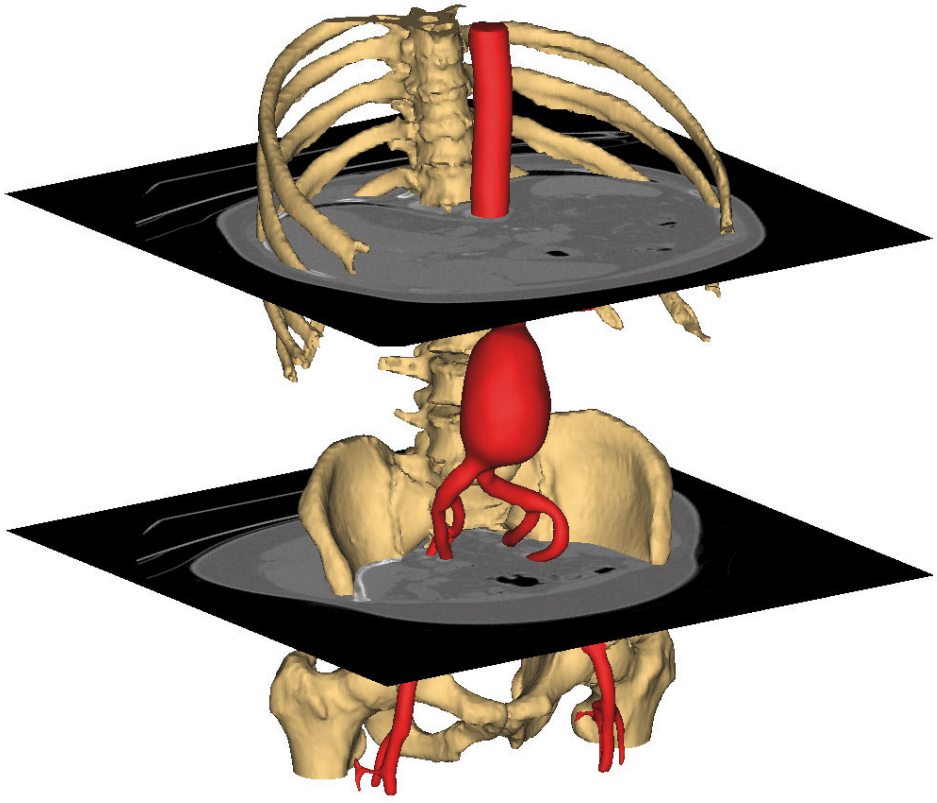

Figura 14. Imagen generada con Mimics con cortes transversales

Los datos en 3D generados por Mimics pueden ser usados en 3-matic para realizar una variedad de operaciones de diseño y mallado directamente sobre los datos anatómicos eliminando así la necesidad de un largo proceso propenso a errores de ingeniería inversa (fig. 15). Esto maximiza la precisión y ahorrara tiempo. Los datos anatómicos o el diseño pueden ser exportados a cualquier CAD y todos los principales paquetes CAE.

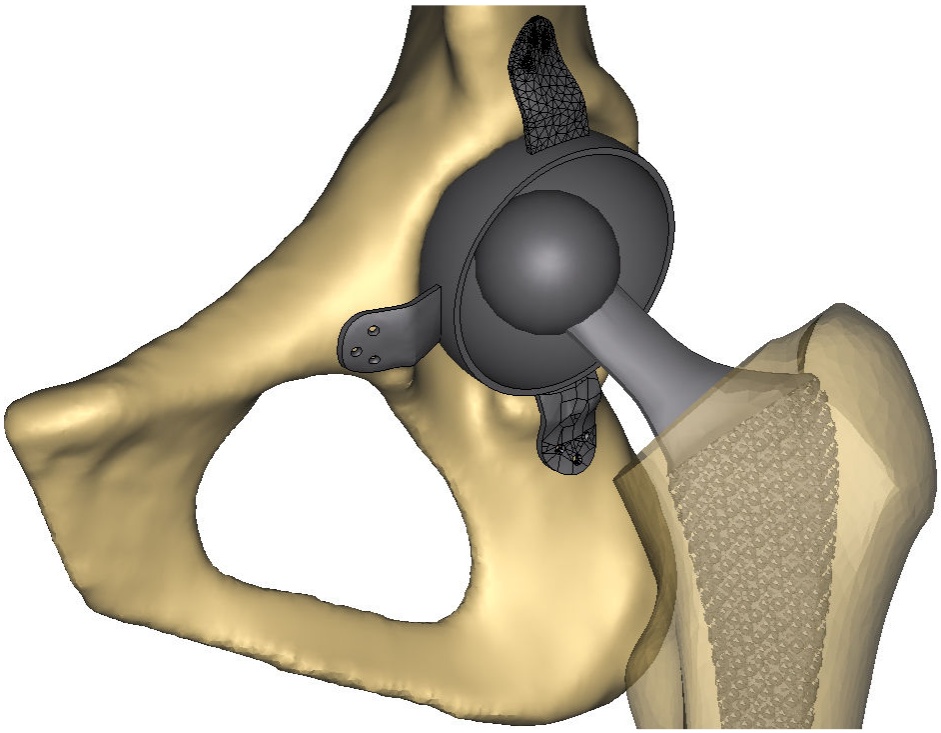

Figura 15. Diseño de una prótesis de cadera sobre una imagen 3D

Finalmente tenemos el conjunto de servicios que está más orientado a la parte de la empresa puesto que consiste en grupos de trabajo divididos en ingeniería, consultoría y desarrollo de software. Estos grupos de trabajo son contratados por otras empresas según sean sus requerimientos.

STSF [20]

La propuesta de esta aplicación es la de proporcionar un entrono de investigación que acelerare el desarrollo y complejidad de los procedimientos quirúrgicos y terapéuticos sin las restricciones que acompañan a los sujetos vivos.

Los investigadores serán capaces de trabajar con imágenes simuladas de tejidos, órganos e instrumental quirúrgico para planificar, desarrollar y evaluar todo el procedimiento quirúrgico mínimamente invasivo, sin el uso de animales o humanos.

Dispone de varios modelos de simulación como: la simulación de imágenes, electrofisiológica, visual, modelado de lesiones, propiedades mecánicos de los tejidos, instrumental quirúrgico e interfaz humana.

Podemos decir que es un paquete bastante completo.

3.3.4- GIMIAS

GIMIAS (Graphical Interface for Medical Image Analysis and Simulation) es un entorno de trabajo orientado y centrado en el análisis, modelado y simulación a partir de imágenes biomédicas. El código abierto con el que está constituido el entorno puede ser extendido mediante el uso de *plugin*s. Actualmente se aplica

en investigación y construcción de prototipos de software clínico en los campos de la imagen y simulación cardíaca, la imagen y simulación angiográfica, neurología y traumatología.

Está pensado para ser explotado por dos tipos de usuarios: los investigadores/personal clínico y los desarrolladores/investigadores. En la siguiente figura (fig. 16) vamos a ver cómo están relacionados con GIMIAS

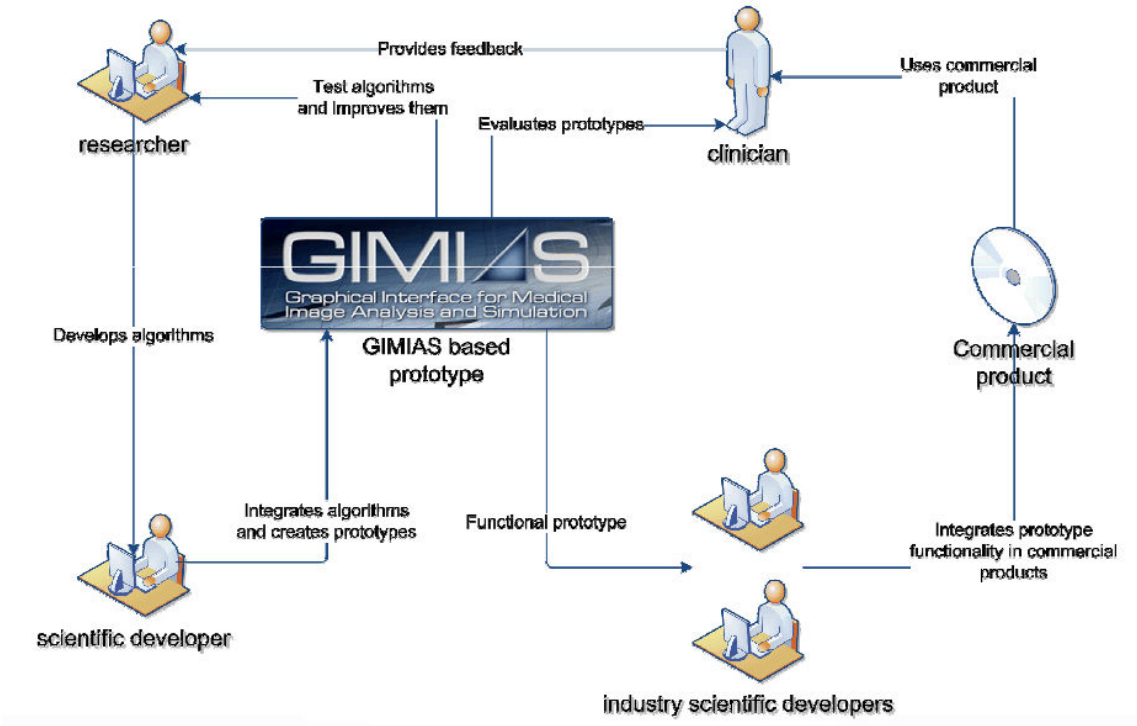

Figura 16. Usuarios de GIMIAS y sus interacciones

Como podemos ver en la imagen, los desarrollos pueden ser comerciales o no dependiendo de quién los realice.

Está desarrollado en C++ con una estructura modular muy definida (fig. 17) y provee un interfaz gráfico para el usuario con todos los datos principales de entrada/salida, funciones de visualización e interacción con imágenes, mallas 3D y señales. Está orientado al diseño basado en *Workflow* (o flujo de trabajo). Para cada aplicación se diseña una cadena de trabajo en la que las diferentes fases se llevan a cabo secuencialmente mediante módulos de software o *plugin*s independientes, que se comunican mediante un espacio de datos común (fig. 18).

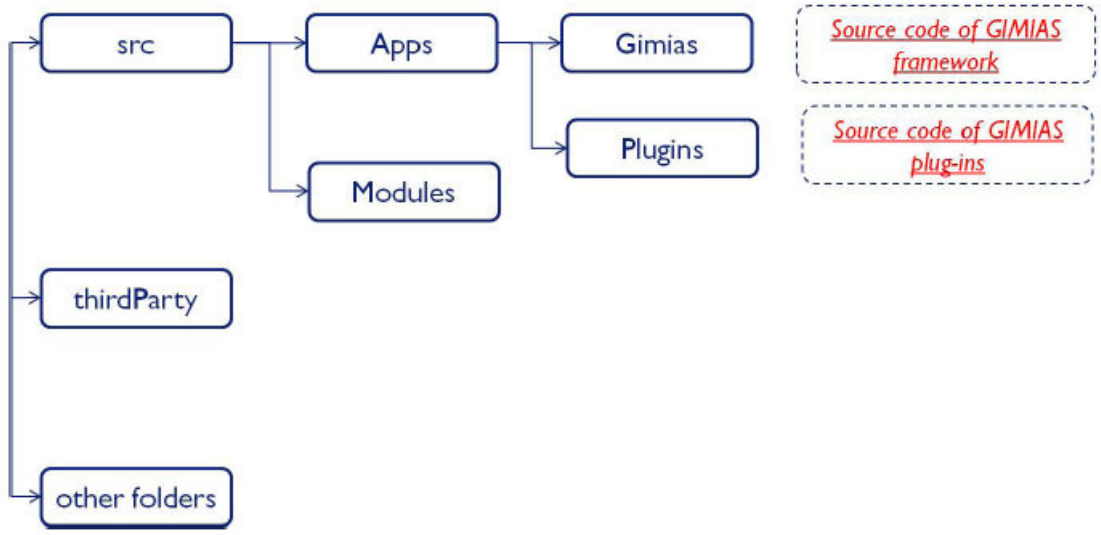

Figura 17. Estructura interna de las carpetas que contienen el código de GIMIAS

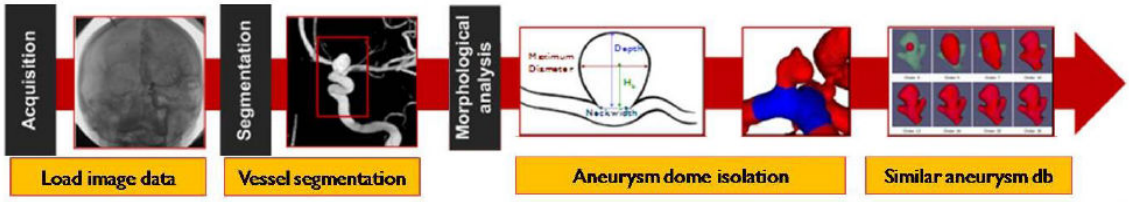

Figura 18. Diagrama de ejemplo de flujo de trabajo de la *AngioMorfología* clinica

A parte de las características ya comentadas anteriormente, también dispone de dos funcionalidades muy importantes para la comunicación. Implementa un estándar de navegación para la transmisión, manejo, almacenamiento e impresión de imágenes médicas conocido como DICOM (referencia) (*Digital Imaging and Comunication in Medicine*) así como un sistema de archivado y comunicación de imágenes conocido como PACS (*Picture Archiving and Communication System*).

Está siendo financiado por diversos proyectos nacionales e internacionales como cvREMOD, euHeart o NoE VPH.

DICOM [21]

Se creó inicialmente en 1985 con la idea de poder decodificar las imágenes generadas por las resonancias magnéticas (RMI) o las tomografías computarizadas (TAC). Rápidamente se constató que necesitaba mejoras.

En 1988 apareció la segunda versión la cual recibió más aceptación por parte de los vendedores. La transmisión de imagen se especificó sobre un par cable 25 dedicado (EIA-485). El primer gran uso de esta tecnología lo realizo el ejercito de los EEUU como parte de MDIS (*Medical Diagnostic Imaging Support*).

En 1993 se volvió a actualizar a su tercera versión. Recibió el nombre de DICOM para aumentar las posibilidades de aceptación a nivel internacional como

un estándar. Se definen nuevas clases de servicio, se le añade soporte de red y se introdujo la declaración de conformidad. Aunque no ha aparecido una versión nueva, se ha ido actualizando y extendiendo desde 1993.

DICOM incluye una definición de formato de archivos y un protocolo de comunicación de red. El protocolo utiliza el estándar TCP/IP para comunicarse entre sistemas. Permite la integración de escáneres, servidores, estaciones de trabajo, impresoras y hardware de red de múltiples fabricantes dentro PACS.

PACS

Es una tecnología de imagen médica que provee almacenamiento económico (y acceso conveniente) para imágenes de múltiples tipos de máquinas. Sirve para transmitir imágenes electrónicas e informes digitalmente eliminando la necesidad de hacerlo de forma manual. Se emplea junto con DICOM puesto que es el formato universal de transmisión. Se constituye de cuatro componentes importantes que son: las modalidades de imágenes (como la tomografía computada de rayos-X e imágenes de resonancia magnética), una red segura para la transmisión de la información de los pacientes, una estación de trabajo (fig. 19) para interpretar y visualizar las imágenes y archivos para el almacenamiento y recuperación de imágenes e informes.

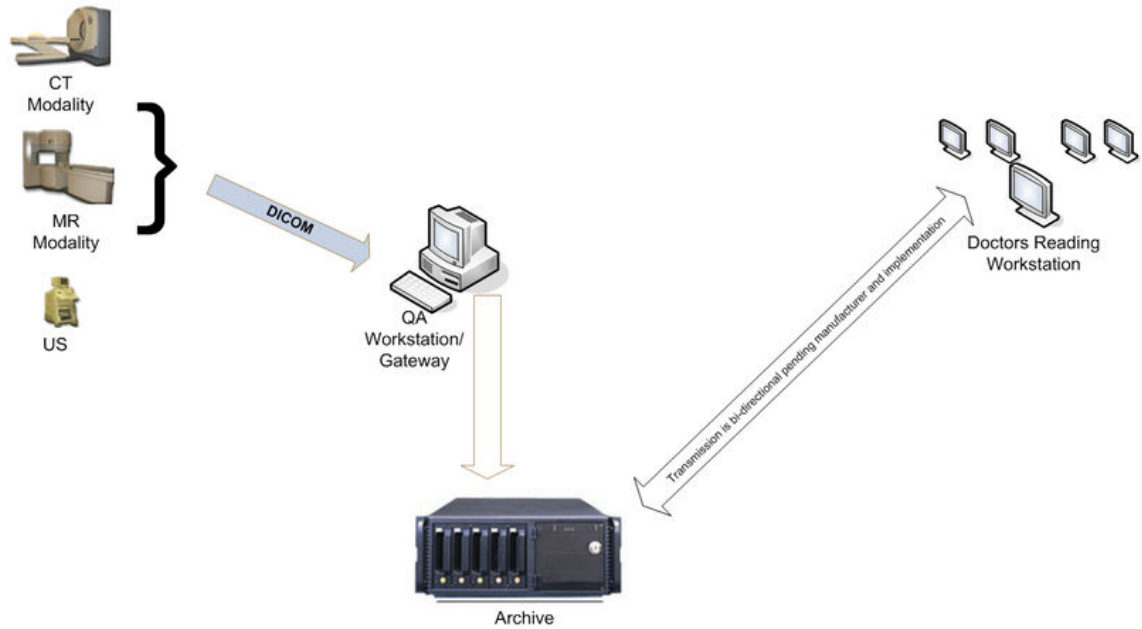

Figura 19. Diagrama de la estación de trabajo de PACS

Reduce las barreras físicas y temporales asociadas a la recuperación, distribución y exhibición de las imágenes basadas en películas/video. Esto se consigue gracias a las nuevas tecnologías que existen actualmente.

Una vez visto los modos de transmisión y como se hace la recuperación de las imágenes, vamos a continuar con las características de GIMIAS.

Permite la visualización de imágenes tanto en 2D como en 3D. Además ofrece la posibilidad de interactuar con ellas. Es multiplataforma, se puede usar

tanto en Windows como en Linux. Se pueden resaltar de forma manual ciertas partes de las imágenes para crear regiones de interés. Tiene una herramienta para crear mallas (incluye refinamiento, corte, orificio de relleno, generación de mallas con volumen, etc…).

Como podemos ver es una herramienta muy completa que nos permite tener a nuestra disposición un gran número de herramientas para realizar las investigaciones pertinentes. Además dispone de métodos para el envió de las imágenes entre dispositivos y/o médicos. Este envío tiene la seguridad necesaria para que se pueda intercambiar la información de los pacientes entre doctores y/o centros. Tiene su propio protocolo de transmisión (DICOM) y almacenamiento de imágenes (PACS).

Aunque hayamos puesto sólo cuatro, existen más paquetes de herramientas para la simulación 3D. Se ha visto un aumento importante a partir del año 2000 en referente a estas herramientas por lo que tienen poco tiempo de vida y aún se están desarrollando nuevas.

Hay herramientas para diferentes simulaciones e investigaciones cuya función consiste en mejorar las pruebas al no tener que depender de sujetos vivos y a su vez permite eliminar una gran parte del coste económico que es necesario durante una investigación médica.

El objetivo de esta memoria no es el de enumerar y revisar todas las aplicaciones existentes, por lo que se ha mostrado un conjunto representativo que muestra la diversidad de usos y magnitud de herramientas existentes. En la bibliografía se incluyen enlaces a diferentes sitios web que completan esta revisión.

4- Especificación

Se pretende implementar un módulo que permita realizar estudios a nivel clínico sobre el corazón, más concretamente sobre la posible relación que pueda existir entre la ya comentada estructura de *Purkinje* (presente en la estructura interna del corazón) y las enfermedades cardiovasculares que puedan derivarse de su mal funcionamiento. Servirá para poder realizar estudios para poder comprender el resultado de diferentes patologías y el propio funcionamiento de la estructura en mayor detalle.

El objetivo es que sea una aplicación totalmente transparente donde el usuario necesite tener conocimientos avanzados, manteniendo la potencia, flexibilidad y la facilidad de uso, y que permita generar un gran abanico de escenarios.

Habrá dos partes dentro de la aplicación. El código de construcción de *Purkinje*, que no será necesario modificar, y la parte del interfaz gráfico que tendrá los menús que hayamos diseñado y que se podría modificar para añadir nuestras propias funcionalidades.

Dado el publico objetivo que mayoritariamente va a utilizar dicha aplicación, queremos que sea lo más intuitiva posible y que el tiempo necesario para aprender a manejarla sea mínimo. Esto se debe a que los usuarios más comunes van a ser médicos e investigadores de los cuales no tenemos garantías sobre su nivel de conocimiento en informática.

4.1- Análisis de requisitos

Vamos a distinguir dentro del proyecto dos tipos de requisitos: los funcionales y los no funcionales. En los requisitos funcionales definimos el comportamiento interno que va a tener la aplicación. En los no funcionales, especificamos criterios que pueden usarse para juzgar la operación de un sistema en lugar de sus comportamientos específicos.

A continuación vamos a exponer y explicar los dos tipos de requisitos explicándolos más detalladamente.

4.1.1- Requisitos funcionales

Los usuarios podrán interactuar con la aplicación mediante una interfaz gráfica en la cual estarán presentes todas las opciones que van a estar disponibles. Todo esto se llevará a cabo mediante la interacción con el ratón.

Las operaciones disponibles en la interfaz gráfica de la aplicación van a ser:

- **Seleccionar una malla**, que será la base sobre la cual se generará la estructura de la red de *Purkinje*.
- **Generar la red de *Purkinje*** según su orden de activación (proximal a distal). Es decir primera, segunda y tercera fase del algoritmo de generación.
- **Seleccionar tres puntos *"atractores"* en la malla**, que hemos elegido primeramente, para después generar la estructura.
- **Generar la red de *Purkinje* alternativa** (distal a proximal). Elegimos tres puntos que serán el final de las tres ramas principales y el inicio de las ramas de la segunda generación.
- **Visualizar por pantalla** la estructura que hemos generado.
- **Importar una estructura** de *Purkinje* que hayamos generado con anterioridad.
- **Exportar la estructura** que acabamos de generar para almacenarla en la memoria por si la necesitamos más adelante.
- **Visualizar por pantalla o en un archivo las estadísticas** referentes a una estructura de *Purkinje* que hemos generado o importado.
- **Posibilidad de interactuar con el modelo 3D** mediante el uso del ratón. Gracias a esto seremos capaces de rotar o hacer zoom.

Dicha interfaz gráfica será un módulo de la aplicación GIMIAS que hemos comentado en el capítulo 2 de este documento. Con esto queremos poder garantizar que pueda ejecutarse en cualquier máquina siempre que se tenga instalado el paquete y el módulo para ejecutar el programa base.

4.1.2- Requisitos no funcionales

Facilidad de uso/Usabilidad

Como hemos comentado anteriormente, nuestro deseo es que la aplicación final sea fácil de usar. Para ello vamos a tener en cuenta las diez reglas de J. Nielsen adaptadas a nuestro sistema:

1. **Visibilidad del estado del sistema:** tendremos todas las posibilidades de que dispone la aplicación a la vista en todo momento gracias a la interfaz gráfica y a sus menús.

2. **Utilizar el lenguaje de los usuarios:** al no conocer exactamente al posible usuario todo se controla con menús intuitivos además de utilizar términos científico-técnicos apropiados.

3. **Control y libertad para el usuario:** el usuario tiene control total sobre la aplicación y todas las operaciones que puede hacer. No hay distinciones ni permisos especiales.

4. **Consistencias y estándares:** gran distinción entre todas las acciones que se pueden realizar y sus nombres para evitar confusiones.

5. **Prevención de errores:** garantizamos que la aplicación sea robusta y consistente pero no que los valores de los parámetros de entrada para realizar las pruebas sean los correctos.

6. **Minimizar la carga de la memoria del usuario:** el usuario no necesita en ningún momento aprenderse de memoria la información que aparece. El encadenamiento de los movimientos a realizar para ejecutar una operación serán mecánicos.

7. **Flexibilidad y eficiencia de uso:** todo se hace de forma gráfica incluso el interactuar con el modelo 3D que hayamos generado por lo que supone que todo se hace de forma rápida y trasparente.

8. **Los diálogos estéticos y diseño minimalista:** se ha diseñado la interfaz gráfica para que contenga la información necesaria para su uso y nada más

9. **Ayudar a los usuarios a reconocer, diagnosticar y recuperarse de los errores:** la aplicación no puede avisar de los posibles errores en los parámetros de entrada puesto que no se disponen de los conocimientos exactos para saber cuándo pueden fallar o no.

10. **Ayuda y documentación:** se pretende que no sea necesario ningún manual de usuario (siempre en la medida de lo posible) para poder exprimir la aplicación hasta su máximo posible.

Accesibilidad

Nos referimos a las capacidades de adaptación que pude disponer nuestra aplicación en cuanto cuestiones de discapacidad.

La mayoría de las acciones que están disponibles en la aplicación pueden ser realizadas con una sola mano. Tanto la generación de la malla como la interacción con la imagen en 3D se llevan a cabo con el ratón puesto que hemos implementado los botones necesarios para que así sea.

En dos casos sería necesario el control del teclado a parte del ratón. Estos serían el ingreso de los parámetros de entrada de la aplicación que se hace mediante un archivo de texto (ya que son numerosos) y el ingreso del nombre con el que se va a guardar el fichero que contendrá la malla generada. Son casos sencillos por lo que no esperamos resulten un inconveniente importante para que el uso generalizado del módulo.

La aplicación es altamente visual e interactiva, tanto en los menús de los casos de uso que tiene la aplicación, como en el resultado que obtenemos (el modelo 3D). Los menús tienen un tamaño estándar para que puedan ser leídos por la gran mayoría de los usuarios de la aplicación. En cuanto al modelo, se le puede realizar zoom tanto para acercarse como para alejarse por lo que puede ayudar para visualizarlo mejor. Por otro lado también se tiene la opción de colorearlo si se desean resaltar ciertos aspectos.

Fiabilidad y Robustez

Al ser un sistema orientado a la investigación, hay que evitar que el usuario se llegue a frustrar debido a los errores propios de la aplicación. De esta manera se tiene que poder diferenciar los errores debidos a la inserción incorrecta de los parámetros como pueden ser inserción de algún carácter no numérico cuando queremos introducir un valor real o que el tipo de dato sea incorrecto. Esto puede llevarse a cabo porque existen una cantidad considerable de parámetros de entrada y estos se escriben en un fichero de texto que posteriormente lee la aplicación. Por lo que hay que controlar esos posibles errores humanos.

Rendimiento

La principal función de la aplicación es la de generar un modelo tridimensional de una red de *Purkinje* mediante los parámetros de entrada que le hayamos puesto. Es por esto que hay que tener en cuenta los recursos que utiliza para que no se llegue a ralentizar demasiado nuestro sistema. El modelo es muy sencillo y no debe de consumir mucha memoria pero es algo que no hay que despreciar. Si se llegase a dar el caso de que se ralentice, la aplicación ya no sería todo lo funcional que quisiésemos por lo que puede llegar a provocar la pérdida de usuarios. La creación tiene que ser lo más rápida posible aunque esto siempre puede variar según la configuración de la computadora donde se genere.

Soporte

La aplicación final debe poner integrarse sin problemas dentro del entorno de prototipado de análisis clínico GIMIAS. Debe ser capaz de ejecutarse en máquinas totalmente diferentes siempre que tengan instalado el sistema general. Podemos decir que nuestro módulo es considerado semi-portable puesto que aunque podamos ejecutarlo en cualquier PC siempre tendremos el requerimiento de GIMIAS. Nuestra aplicación es libre por lo que sería posible añadir nuevas funcionalidades o modificar las ya existentes. El único inconveniente para poder realizar las modificaciones pertinentes es que sería necesario tener un nivel previo medio-alto en conocimiento de programación.

En el siguiente punto vamos a estudiar cuales serian los costes del proyecto tanto a nivel económico como temporal.

4.2- Especificaciones del sistema

A continuación vamos a comentar ciertos aspectos que se han tenido en cuenta a la hora de programar nuestra aplicación final, referentes a la selección del lenguaje de programación y de la librería de gráficos 3D. No se ha incluido este apartado en el estado del arte puesto que ambas elecciones eran un requisito específico impuesto por el cliente por lo que no era necesario compararlos con otras opciones.

4.2.1- Lenguaje de programación

Para la implementación de la aplicación como de la interfaz gráfica, el cliente nos dijo que debía realizarse en C++. Es un lenguaje muy conocido actualmente y más para los lectores de este documento por lo que no se considera necesario dar detalles.

Se incluye documentación necesaria para que sea posible entender y extender la funcionalidad del código desarrollado en este proyecto.

4.2.2- Librería de gráficos 3D: VTK

Dentro de GIMIAS, podemos contar con un amplio abanico de librerías externas o *ThirdParty* para su uso en la posible creación de módulos adaptables. Dependiendo de lo que queramos crear o estemos buscando, podremos utilizar una u otra o varias a la vez. Son completamente *OpenSource* por lo que el coste para la persona que la utilice será de 0€. Es algo de agradecer puesto que de esta manera se deja abierta la creación de aplicaciones diferentes y sólo dependerá de la habilidad del usuario el generarlas.

Nuestra aplicación está orientada a la generación de un modelo en 3D de una estructura de red de *Purkinje*. Por ello nosotros vamos a hacer uso de la librería gratuita VTK (*visualization toolkit*) [22] creada en C++ y orientada a la visualización y creación de gráficos 3D. A continuación vamos a explicar con más detalle en qué consiste dicha librería, sus orígenes y como funciona.

Librería ThirdParty: VTK

Actualmente hay dos tendencias importantes que predominan dentro de la industria informática: el desarrollo de sistemas orientados a objetos y el uso de métodos más complejos para crear interfaces de usuario, en especial el uso de gráficos en 3D.

Estas dos tendencias ofrecen al profesional (la programación orientada a objetos) y al usuario (los gráficos 3D), una serie de ventajas a tener en cuenta. Los sistemas orientados a objetos, ofrecen la posibilidad de crear unos sistemas mejores, más fácil de mantener con software reutilizable. Por otro lado, la infografía ofrece una ventana al equipo y a los mundos virtuales creados ahí y cuando se junta con la visualización 3D permite que los usuarios puedan explorar rápidamente y comprender los sistemas complejos como es el caso de nuestra red de *Purkinje*. Como siempre se ha dicho más vale una imagen a mil palabras.

VTK, cuyo origen fue sobre el año 1993 (Ref. anexo I pt 1), es una librería *OpenSource* disponible para su uso en gráficos 3D por ordenador (Ref. anexo I pt 2), modelado, procesamiento de imágenes, representación volumétrica, visualización científica y de información. VTK también incluye soporte auxiliar para *widget*s interactivos en 3D, anotaciones en 2D y 3D y computación paralela. El núcleo de VTK esta implementado como un conjunto de herramientas en C++ requiriendo a la hora de crear aplicaciones que los usuarios combinen varios objetos. El sistema también soporta el ajuste automático de C++ a Python, Java y

Tcl por lo que para crear nuestras aplicaciones también podemos emplear cualquiera de estos lenguajes de programación interpretados (Ref. anexo I pt 3).

VTK emplea el Software de Calidad de Procesos Kitware (Cmake, CTest, CDash y CPack) para construir, probar y empaquetar el sistema. Esto consigue que VTK sea una aplicación de desarrollo multiplataforma basada en pruebas y la programación extrema que permite a la aplicación generar código robusto y de alta calidad. Esta librería se utiliza en todo el mundo para aplicaciones comerciales, de investigación y desarrollo y es la base de multitud de aplicaciones de visualización avanzadas, tales como: ParaView, Visita, VisTrails, Slicer, Mâyâvi y OsiriX.

Como punto importante cabe destacar que esta librería está diseñada para soporta el multiprocesamiento (Ref. anexo I pt 4).

Gracias a esta propiedad, podríamos generar datos de un mismo tipo a la vez con lo que podríamos procesar un gran *dataset* particionándolo en conjuntos inferiores y posteriormente calculando al mismo tiempo cada conjunto.

Esto permite un gran avance en cuanto al tiempo que requeriríamos para generar nuestras soluciones si fuese muy pesada. Podríamos migrar toda la aplicación a máquinas con más de un procesador si viésemos que lo necesitamos.

VTK dispone de un modelo de visualización con estilo de flujo de datos (Ref. anexo I pt 5). Gracias a esto, podemos realizar series de cálculos aritméticos sobre un valor de entrada y recapturarlo a la salida del sistema. A la hora de visualizar los datos, disponemos de cinco tipos diferentes: *vtkStructuredPoints*, *vtkPointSet*, *vtkStructuredGrid*, *vtkPolyData* y *vtkUnstructuredGrid*. Pero esto puede tener un problema añadido en cuanto a la gestión de la memoria por lo que se realizan ciertos cambios para resolverlo como por ejemplo elegir si queremos predominar la memoria sobre los cálculos o al revés (Ref. anexo I pt 6).

Para una correcta visualización y que no produzca errores con los tipos de los datos de entrada o con posibles bucles infinitos, se constituye una red de visualización que pretende eliminar estos problemas y tenga una correcta ejecución (Ref. anexo I pt 7).

El diseño orientado a objetos que constituye el *toolkit* VTK es el poder tomar ventaja de la jerarquía de herencia del *dataset* para construir objetos de proceso genéricos o específicos.

Convenios

Para un mejor uso de las posibilidades que ofrece el *toolkit* y para facilitar en la medida de lo posible al usuario la creación de aplicaciones, se decidieron una serie de convenios. Algunos son simples como el uso de un esquema de nombres estándar, largo y descriptivo para los objeto, métodos y variables; adoptando plantillas estándar y estilos de codificación; aplicando un prefijo “vtk” a los nombre de los objetos para impedir colisiones de espacios de nombre con

otras librerías de clases de C++. Aunque los dos convenios más importantes que se han tomado han sido, la incorporación de la documentación directamente en el código y el uso de los métodos estándar *Set/Get* para leer y escribir variables de instancia de objetos.

La inclusión de la documentación en el código tiene dos funciones importantes. La primera permite a los desarrolladores el obtener vía on-line los archivos fuente y las cabeceras para conseguir información sobre objetos específicos. La segunda es la de permitir a los creadores del *toolkit* el generar de forma automática páginas y documentación web en HTML. Con esto se consigue que todo vaya junto por lo que sólo hay que consultar un fichero y no dos. Para la lectura y establecimiento del valor de las variables de objetos se usa la denominación estándar *Get* y *Set*. Cuando se establece el valor de una variable de instancia, se compara los valores antiguos con los nuevos y sólo se modifica el tiempo de actualización del objeto si el valor de la variable instancia ha cambiado.

Con todo esto nos podemos hacer una idea más clara de la estructura de VTK. Como podemos ver prima la portabilidad y la sencillez de uso dos de las ideas principales de punto de partida para la creación del *toolkit*. Sirve tanto para generar objetos en 3D como para visualizarlos. Junta las ventajas de los lenguajes compilados con los interpretados. Es un conjunto de herramientas muy completas, robustas y que parte con ideas muy interesantes. Además hay que añadir que es *OpenSource* por lo que detrás tenemos una gran comunidad de usuarios que pueden ir mejorándolo y dando apoyo a otros usuarios por lo que por ahora tiene el futuro garantizado.

4.3- Planificación y estimación de costes

En este apartado vamos a estudiar la viabilidad que tiene el proyecto mediante la realización de los cálculos para determinar la estimación de coste (esfuerzo, económico y temporal). Primeramente utilizaremos el método de Puntos de Casos de Uso para conocer la estimación de la duración temporal (en meses).

A continuación realizaremos la subdivisión de tareas necesarias para afrontar con éxito el proyecto. Para terminar, y ya conociendo los otros dos cálculos anteriores, realizaremos un presupuesto a medida para el cliente.

4.3.1- Cálculo del tiempo

Vamos a emplear la técnica del cálculo del coste temporal dado por el cálculo de Puntos de Caso de Uso [23]. Este método da unos resultados muy parecidos a los de COCOMO II y consideramos que su método se ajusta más a nuestro proyecto. En el anexo III se explicará más detalladamente dicho método.

Cálculo de los Puntos de Casos de Uso sin Ajustar:

UUCP = UAW + UUCW *(1)*

Donde: **UUCP:** Puntos de casos de uso sin ajustar.
UAW: Factor de peso de los actores sin ajustar.
UUCW: Factor de peso de los casos de uso sin ajustar.

Para el cálculo de UAW nos basaremos en la siguiente tabla:

Tipo de actor	Descripción	Factor de peso
Simple	Otro sistema que interactúa con el sistema a desarrollar mediante una interfaz de programación	1
Medio	Otro sistema que interactúa con el sistema da desarrollar mediante un protocolo o una interfaz basada en texto	2
Complejo	Una persona que interactúa con el sistema mediante una interfaz gráfica	3

Tabla 1. Factor de peso de los actores sin ajustar

En nuestros casos de uso sólo existe un actor y será el usuario final (es decir una persona física) por lo que:

UAW = 1 x 3 = 3 *(2)*

La tabla a continuación nos ayudará a establecer el valor de UUCW:

Tipo de caso de uso	Descripción	Factor de peso
Simple	El caso de uso contiene de 1 a 3 transacciones	5
Medio	El caso de uso contiene de 4 a 7 transacciones	10
Complejo	El caso de uso contiene más de 8 transacciones	15

Tabla 2. Factor de peso de los casos de uso sin ajustar

Disponemos de un total de 8 casos de uso y todos ellos tienen de 1 a 3 transacciones lo que nos da como resultado:

UUCW = 8 x 5 = 40 *(3)*

Dándonos como resultado final:

UUCP = 3 + 40 = 43 *(4)*

Una vez calculado los Puntos de Casos de Uso sin Ajustar, procedemos a determinar los Puntos de Casos de Uso Ajustados.

Cálculo de los puntos de casos de uso ajustados:

UCP = UUCP x TCF x EF *(5)*

Donde: **UCP:** Puntos de casos de uso ajustados.
UUCP: Puntos de casos sin ajustar.
TCF: Factor de complejidad técnica.
EF: Factor de ambiente.

Como el valor de UUCP ya lo hemos calculado anteriormente, pasaremos a calcular el Factor de Complejidad Técnica (o TCF).

Factor	Descripción	Peso	Valor asignado	Comentario
T1	Sistema distribuido	2	0	El sistema es centralizado
T2	Objetivos de performance o tiempo de respuesta	1	3	La velocidad de la aplicación debe ser aceptable
T3	Eficiencia del usuario final	1	1	Escasas restricciones de eficiencia
T4	Procesamiento interno complejo	1	3	Hay cálculos complejos
T5	El código debe de ser reutilizable	1	4	Se requiere bastante reutilización del código
T6	Facilidad de instalación	0.5	1	Va integrado en un módulo más general
T7	Facilidad de uso	0.5	3	Normal
T8	Portabilidad	2	0	No se requiere que el sistema sea portable puesto que ya va integrado
T9	Facilidad de cambio	1	1	Se requiere un costo muy bajo de mantenimiento
T10	Concurrencia	1	0	No hay concurrencia
T11	Incluye objetivos especiales de de seguridad	1	0	No es necesaria ninguna seguridad
T12	Provee acceso directo a terceras partes	1	0	Solo puede entrar 1 usuario a la vez
T13	Se requieren facilidades especiales de entrenamiento a usuarios	1	2	Sistema fácil de usar

Tabla 3. Factor de complejidad técnica

Con los valores de la tabla anterior, el valor de TCF será:

TCF = 0.6 + 0.01 x ∑(Peso$_i$ x Valor asignado$_i$) *(6)*

Entonces:

TCF = 0.6 + 0.01 x (2x0 + 1x3 + 1x1 + 1x3 + 1x4 + 0.5x1 + 0.5x3 + 2x0 +
+ 1x1 + 1x0 + 1x0 + 1x0 + 1x2) *(7)*

TCF = 0.6 + 0.01 x 16 = 0.76 *(8)*

A continuación hay que calcular el valor del Factor Ambiente (EF):

Factor	Descripción	Peso	Valor asignado	Comentario
E1	Familiaridad con el modelo de proyecto utilizado	1.5	2	Se conoce un poco el modelo de proyecto
E2	Experiencia en la aplicación	0.5	0	No se tiene ninguna experiencia en la aplicación
E3	Experiencia en orientación a objetos	1	5	Se dispone de amplios conocimientos
E4	Capacidad del analista líder	0.5	5	Experiencia altamente demostrada
E5	Motivación	1	5	Muy elevada
E6	Estabilidad en los requerimientos	2	2	Se esperan pequeños cambios
E7	Personal part-time	-1	0	Todo el mundo es full-time
E8	Dificultad del lenguaje de programación	-1	1	Se usará el lenguaje C++

Tabla 4. Factores de ambiente

EF = 1.4 – 0.03 x ∑(Peso$_i$ x Valor asignado$_i$) *(9)*

Esta fórmula nos da:

EF = 1.4 – 0.03 x (1.5x2 + 0.5x0 + 1x5 + 0.5x5 + 1x5 + 2x2 + -1x0 +-1x1)

EF = 1.4 – 0.03 x 18.5 = 0.845 *(11)*

Una vez calculados los valores TCF, EF y teniendo el valor de UUCP, podemos calcular el valor final de UCP como:

UCP = 43 x 0.76 x 0.845 = 27.61 *(12)*

Ahora mismo tenemos el resultado final de los Puntos de Casos de Uso Ajustados. Pero para conocer el valor temporal, tenemos que transformar dicho valor en esfuerzo. Para ello, y según la explicación dada en el anexo, tomaremos

que cada Punto de Casos de Uso requiere 20 horas-hombre. Esto nos dará un valor de Esfuerzo:

E = UCP x CF *(13)*

Donde: **E:** Esfuerzo estimado en horas-hombre.
UCP: Puntos de Casos de Uso Ajustados.
CF: Factor de conversión.

E= 27.61 x 20 = 552.2 *(14)*

Obtenemos un valor de 552,2 horas-hombre. Esto nos da la estimación del esfuerzo en horas-hombre contemplando sólo el desarrollo de la funcionalidad especificada en los casos de uso. Por ello para calcular una estimación más precisa vamos a determinar las partes correspondientes a las diferentes actividades relacionadas con la elaboración del software. Dicha estimación se realizara en base a unos porcentajes determinados. Hay que tener en cuenta que estos valores no son absolutos y pueden variar según se vayan realizando.

Actividad	Porcentaje	Horas-Hombre
Análisis	10%	138.05
Diseño	20%	276.1
Programación	40%	552.2
Pruebas	15%	207.075
Sobrecarga	15%	207.075
Total	100%	1380.5

Tabla 5. Proporciones temporales

Si traducimos este valor a meses nos dará (teniendo en cuenta que un mes tiene aproximadamente 160 horas):

Tiempo (en meses) = 1380.5 / 160 = 8.6 meses *(15)*

En conclusión para la realización de dicho proyecto, será necesario unos 8 meses y medio.

4.3.2- Planificación temporal

En primer lugar vamos a exponer las diferentes fases de nuestro proyecto para tener una idea rápida del posible coste temporal que puede tener.

A continuación podremos realizar un diagrama que contenga los tiempos ya definidos con la idea de estructurarlo de tal forma que tengamos la mejor planificación posible para organizarnos mejor.

Hemos dividido nuestro proyecto en 11 tareas principales en las cuales existen diversas subtareas.

1. Toma de contacto
2. Comprensión de requisitos

3. Búsqueda de información
4. Código aplicación I
5. Código aplicación II
6. Interfaz gráfica
7. Testeo
8. Memoria
9. Documentación
10. Relectura final
11. Presentación PowerPoint

Gracias al diagrama de Gantt podemos tener, de forma más visual, las tareas y como están estructuradas. De esta manera podemos llegar a organizarnos mejor y conseguimos no malgastar el tiempo. Hay que tener en cuenta que dicha división temporal es orientativa y puede variar al durante todo el proceso.

Por otra parte hay que tener en cuenta que dicho proyecto lo va a realizar una sola persona por lo que en raros casos se podrá ir adelantando tareas sin antes haber terminado la anterior.

Nombre	Fecha de inicio	Fecha de fin
T1: Toma de contacto	4/10/10	5/10/10
T2: Comprension de r...	5/10/10	8/10/10
T3: Busqueda de infor...	8/10/10	31/10/10
T3.1: El corazón	8/10/10	31/10/10
T3.2: GIMIAS	8/10/10	31/10/10
T4: Código aplicación I	2/11/10	1/12/10
T4.1: Diagrama d...	2/11/10	4/11/10
T4.2: Diagrama d...	4/11/10	14/11/10
T4.3: Diagrama d...	15/11/10	1/12/10
T5: Código aplicación II	2/12/10	3/02/11
T5.1: Estudio del ...	2/12/10	23/12/10
T5.2: Implementa...	11/12/10	3/02/11
T6: Interfaz gráfica	3/02/11	27/04/11
T6.1: Estudio req...	12/02/11	17/02/11
T6.2: Diseño	17/02/11	3/03/11
T6.3: Implementa...	3/02/11	27/04/11
T7: Testeo	27/04/11	22/05/11
T8: Memoria	23/12/10	31/07/11
T9: Documentación	8/10/10	16/04/11
T10: Relectura final	1/09/11	8/09/11
T11: Presentación po...	12/09/11	20/09/11

Figura 20. Tareas y subtareas

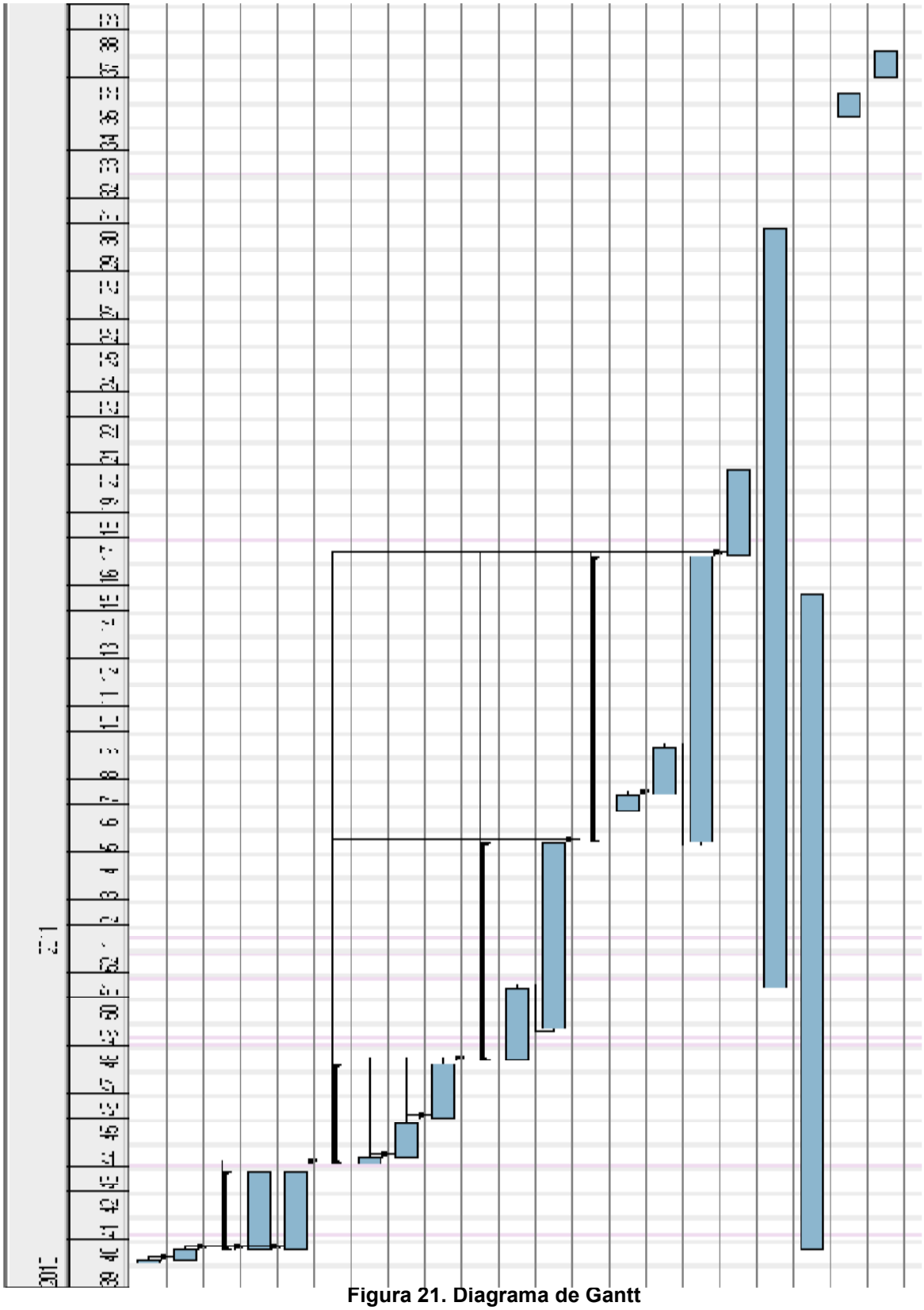

Figura 21. Diagrama de Gantt

4.3.3- Estimación económica

Una vez establecido el coste temporal del proyecto, vamos a calcular cual sería su coste económico.

Coste hardware

Para la elaboración del proyecto se ha adquirido un PC portátil Dell Precision nuevo cuyo coste ha sido de 1566€. Esta adquisición no formará parte del producto final puesto que no se le va a entregar al cliente. Por ello vamos a realizar una estimación de la amortización suponiendo que la vida útil a pleno rendimiento y sin verse afectado en un ordenador portátil es de 7 años.

Amortización: 1566 * 1/7 = 223,71€ *(16)*

Coste software

La aplicación está basada en la idea *OpenSource* para que sea asequible para cualquier investigador por lo que el coste de la plataforma GIMIAS es de 0€. Para la realización de la memoria y la posterior presentación se va a emplear el paquete Office que te viene ya con el PC. En cuanto al programa de implementación vamos a utilizar el Microsoft Visual Studio 2008 por requerimiento de ciertas aplicaciones. Hemos encontrado un distribuidor web con muy buen precio por lo que podríamos conseguirlo por 99$ que al cambio en € serían 69,20€. También tuve que ir a Barcelona a realizar un curso para aprender a programar la aplicación. El curso era gratuito pero el viaje en tren no, su coste subió a 60€.

Resumiendo coste hardware / software:

Descripción	Precio (en €)
Ordenador portátil (con Windows)	0,00
Paquete GIMIAS	0,00
Microsoft Office Starter	0,00
Viaje+Curso Barcelona	60,00
Microsoft Visual Studio 2008	69,20

Tabla 6. Resumen coste hardware/software

Coste personal

Como hemos visto en la estimación temporal que hemos hecho, el proyecto se realizará en un periodo de tiempo de unos 8 meses y medio aproximadamente. Según el cálculo del coste final horas-hombre, el proyecto se realizará en 1380.5 horas. Considerando la categoría profesional de Ingeniero Júnior, cuyo salario es de 17€ la hora, obtendríamos el siguiente coste:

1380.5 * 17 = 23.468,50 € *(17)*

Nuestro proyecto al ser una herramienta *OpenSource*, no va a ser comercializado por lo que el gasto total ascenderá a nuestros honorarios más los gastos acarreados del material y aprendizaje.

Calculo de los honorarios totales:

En este punto entran las contingencias (4%) acarreadas por los gastos que puedan ir apareciendo a lo largo del periodo de realización del proyecto (como reuniones canceladas, materiales necesarios, etc…). Teniendo en cuenta esto, los honorarios totales ascenderán a:

23.468,50 x 1,2 x 1,04 = 29.184,69 € *(18)*

A este valor hay que añadirle los gastos de material (con su correspondiente partida de contingencia del 5%) y posibles cursos con lo que los honorarios totales serán:

29.184,69 + (223,71 x 1,05) + 60 + (69,20 x 1/6) = 29.491,12 € *(19)*

Finalmente nuestros honorarios ascenderán a **29.491,12 Euros**.

4.3.4- Viabilidad del proyecto

A continuación vamos a estudiar la viabilidad general del proyecto (económica, técnica, legal e impacto distributivo).

Viabilidad económica

En el apartado anterior, hemos realizado el cálculo del coste económico del proyecto. Como podemos ver, se han calculado los márgenes de beneficio sobre el coste total, mostrando la rentabilidad del proyecto en relación con las horas invertidas y el beneficio que se obtiene de él.

En cuanto al cliente, como hemos expuesto anteriormente, este es un proyecto de software orientado a la investigación por lo que el coste se tiene que ajustar lo más posible, aunque dependiendo de las partidas económicas que dedique la empresa a su área de I+D+i no tendría por qué considerarse una limitación.

Viabilidad técnica

Vistos los requisitos funcionales que hemos ido exponiendo, podemos decir que actualmente la tecnología existente en el campo de la informática nos permite abordar este proyecto desde el punto de vista de los requisitos tecnológicos. Los ordenadores existentes actualmente suelen llevar como mínimo dos núcleos y un procesador gráfico acorde a las necesidades 3D que se requieren. El lenguaje de programación empleado es de código abierto por lo que tampoco tendríamos problemas.

En lo referente a los recursos humanos, sólo se cuenta con una persona puesto que se trata de un proyecto final de carrera con lo que es el estudiante el que se tiene que encargar de realizar todas las tareas y terminarlas en su debido plazo de tiempo.

Viabilidad legal

Al tratarse de un software de investigación, hay que facilitar en la medida de lo posible los datos biológico/médicos que se conozcan para poder ejecutar la aplicación. Al estar más orientada a la investigación universitaria y al pertenecer a un grupo de trabajo a nivel internacional, no pensamos que puedan existir problemas en cuanto a la comunicación de información entre diferentes grupos de investigadores, más allá de las restricciones de proyectos.

Si por el contrario se usase en la investigación privada, nosotros no nos comprometemos a realizar un almacenamiento y proteger los datos, eso lo dejamos en manos de la empresa responsable.

Impacto distributivo

El software que estamos elaborando puede tener un gran impacto a nivel internacional puesto que, aparte de constituir parte de un software mayor, y tener detrás un grupo de investigación muy importante. Puede ayudar a estudiar uno de los grandes problemas de la sociedad actual. Es por ello que su distribución puede ser muy importante dentro del marco de la investigación cardiovascular puesto que se puede llegar a adaptar a diferentes investigaciones sobre el corazón.

5- Desarrollo del proyecto

En este capítulo, vamos explicar las diferentes fases que ha seguido el proyecto durante su desarrollo. Empezaremos haciendo un análisis de su estructura mediante un conjunto de diagramas descriptivos. A continuación, comentaremos el diseño centrándonos en la explicación de los casos de uso que hayamos presentado antes, y mostraremos cómo será la interfaz gráfica. Para finalizar, hablaremos de la implementación de la aplicación tanto de las clases como de la propia interfaz.

5.1- Diagramas

Vamos a estudiar la aplicación mediante sus diagramas para tener una visualización más gráfica de cómo está constituida.

5.1.1- Diagrama de casos de uso

Nuestro sistema ha de ser sencillo de utilizar y orientado a un abanico de usuarios con una gran variedad de perfiles técnicos. Todos los perfiles han de tener acceso a toda la funcionalidad. Con esto queremos asegurarnos que cualquiera que pueda hacer pruebas las haga sin problemas. En resumen, sólo tenemos un tipo de usuario que pueda acceder a la aplicación por lo que sólo tendremos un único diagrama de casos de uso. A continuación vamos a ver como es (fig. 22).

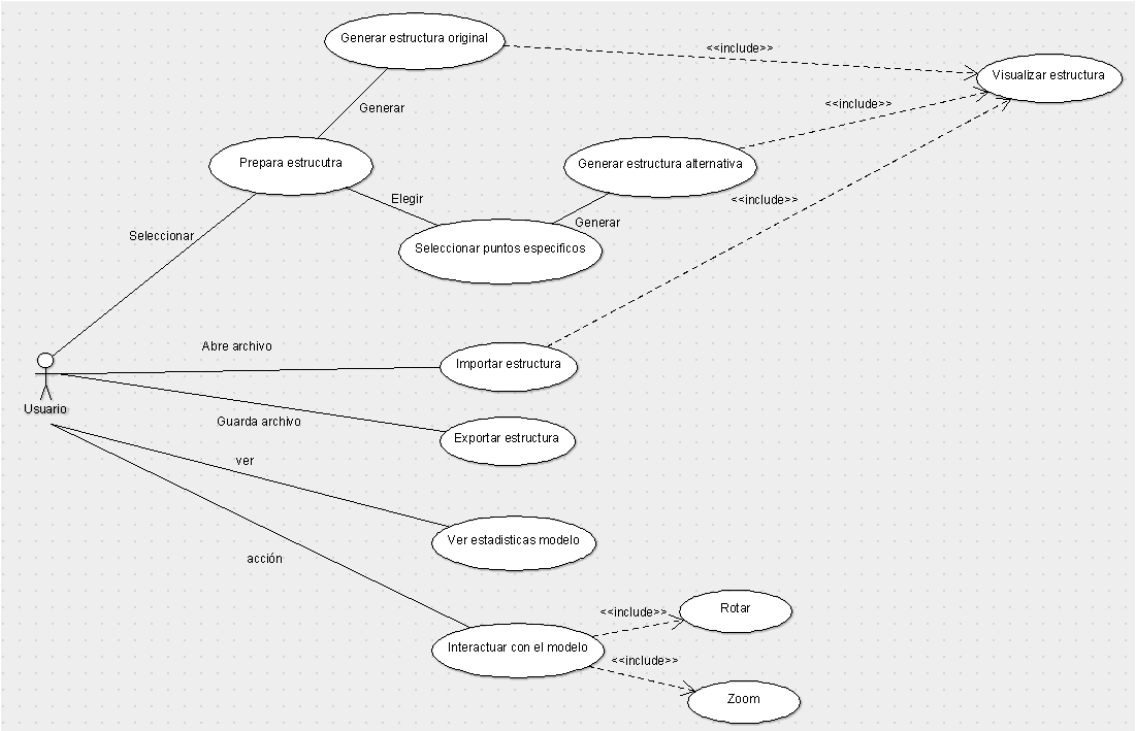

Figura 22. Diagrama de casos de uso del sistema.

Como podemos ver, un único tipo de usuario puede realizar todas las acciones posibles de la aplicación.

Tenemos dos maneras diferentes de generar nuestra estructura de la red de *Purkinje*. La manera original que consiste en generar la estructura de manera que las tres ramas principales siempre terminan en la misma región y la otra que genera la red alternativa donde elegimos esos puntos de finalización. Pensamos que esto puede resultar útil para las investigaciones posteriores puesto que podemos presentar dos estilos diferentes de generación y así comparar los resultados obtenidos. Luego tenemos las funciones clásicas importar/exportar, muy necesarias en este caso como lo es también la opción de ver estadísticas del modelo. Para terminar con los más importantes, tenemos el caso de uso de interactuar con el modelo. Esto nos permitirá hacer zoom en la imagen o rotarla para verla desde diferentes puntos.

Los casos de uso que no hemos citado son intermediarios y se detallarán más adelante de forma precisa.

5.1.2- Diagrama de clases

El uso del lenguaje de programación C++ nos ha permitido estructurar nuestro sistema en bloques y tener una organización basada en clases.

Hemos creado tres bloques principales que van a interactuar entre sí para que se pueda ejecutar nuestra aplicación. A continuación describiremos cada bloque de forma individual, y finalmente mostraremos un diagrama sencillo para explicar cómo se interrelacionan.

El primer gran bloque (o bloque A) va a constituir las clases principales que hemos creado para que nuestra aplicación se comporte como deseamos.

En dicho bloque tendremos cinco clases (csPkjT, csPkj2T, csPkjTime, csStructUtils y csSurfaces). Cada una sirve para una determinada parte de la aplicación que explicaremos más detalladamente en el punto de la implementación. Tenemos dos clases básicas: csPkjT y csPkjT2. La primera se encarga de generar la primera fase de la estructura. La segunda sirve para generar la segunda y tercera fase. Tres clases adicionales se usan internamente, pues contienen fórmulas matemáticas para el cálculo y opciones para el tratamiento de la superficie antes de generar la estructura o incluso el cómputo del tiempo que se tarda en generarla.

Vamos a definir la asociatividad que tienen las diferentes clases entre sí. csPkjT y csPkj2T tiene un grado de asociación uno entre ellas. Si se genera una por fuerza se tiene que generar la otra, no pueden ser independientes puesto que sino la aplicación no funcionaría como deseamos.

csPkjT también se relaciona con las otras tres clases (las que sirven de apoyo) con una asociación de 0...* en el lado de las clases de apoyo (csStructUtils y csSurfs), salvo csPkjTime que será 1 y 1 en el lado de csPkjT. Esto se debe a que al generar la primera fase, se pueden generar (y además se generan) diferentes objetos de las clases de apoyo pero estas a su vez sólo pueden ser llamadas por un objeto de la clase csPkjT a la vez. El tema de la clase que contiene los métodos para calcular el tiempo sólo se genera un objeto que es el que estará ejecutándose mientras se vaya generando la red de *Purkinje*.

Lo mismo pasa con csPkj2T. Tenemos 0…* tanto en el lado de csStructUtils como en el de csSurfs y 1 en el de csPkjTime. A su vez sólo podrán interactuar con un objeto de la clase csPkj2T por lo que la asociación al otro lado será de 1.

Con este bloque conseguimos reunir las clases que van a calcular los datos necesarios para generar la estructura.

El siguiente bloque (o bloque B) es el de la librería gráfica VTK.

Anteriormente hemos detallado como está constituida la librería. Esta contiene un gran número de clases por lo que alargaría enormemente la memoria y nos alejaríamos demasiado del tema central por lo que si se desea saber más recomendamos la visita a la dirección web que se detalla en la referencia.

Por último, el tercer bloque (o bloque C) constituye la clase que contiene la interfaz gráfica y sus múltiples *widget*s.

Este bloque está formado por la clase que va a controlar los *widget*s que nos permitirán realizar todos los casos de uso que hemos expuesto anteriormente. Aquí se implementará el código para dar forma a dichos menús. Es una clase muy importante para el correcto desarrollo de nuestra aplicación.

Como última puntualización en lo que respecta a nuestro diagrama de clases, vamos a ver como se relacionan entre sí los tres bloques.

La asociación entre el bloque A y el B es de 1…* en el lado de la *ThirdParty* a 1 en el otro lado. Esto significa que el bloque A puede usar múltiples objetos de la librería gráfica pero estos objetos sólo pueden estar asociados a un bloque A al mismo tiempo. En lo referente a la relación entre el bloque A y el bloque B es de 1 a 1 puesto en la interfaz gráfica sólo se puede ejecutar una sola red de *Purkinje* a la vez.

Como ya hemos comentado, más adelante entraremos en detalles, ahora lo que hemos hecho ha sido una explicación ligera para que se tenga una idea de cómo está estructurado nuestro sistema. A continuación vamos a ver los diagramas de secuencia de los casos de uso.

5.1.3 Diagramas de secuencia

En este apartado vamos a explicar con la ayuda de los diagramas de secuencias, que acciones se van a llevar a cabo cuando se ejecuten los diferentes casos de uso del sistema.

Consideramos que el entendimiento de alguno de ellos puede resultar un poco complicado por eso los diagramas son tan detallados. Aquí pondremos los más generales y en el apartado de implementación, donde se explica mejor que hace cada método de la clase, pondremos los más detallados para poder consultarlos mejor.

Generar estructura original:

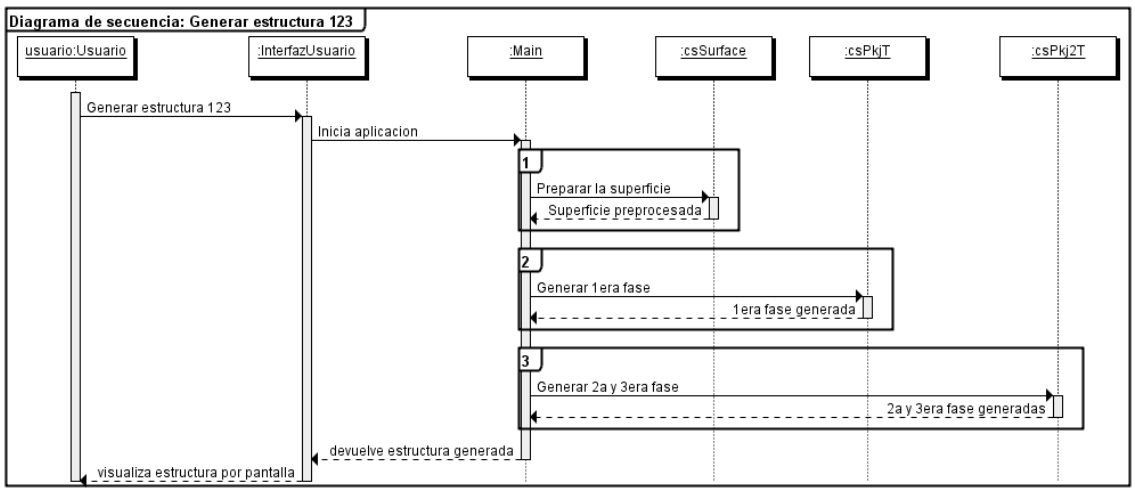

Figura 23. Diagrama de secuencia para "Generar estructura 123"

La activación la realiza el usuario a través de la interfaz gráfica. Como vemos la aplicación primero prepara la estructura para después generar la red de *Purkinje* por fases. Cuando finaliza se muestra por pantalla el resultado.

Generar estructura alternativa:

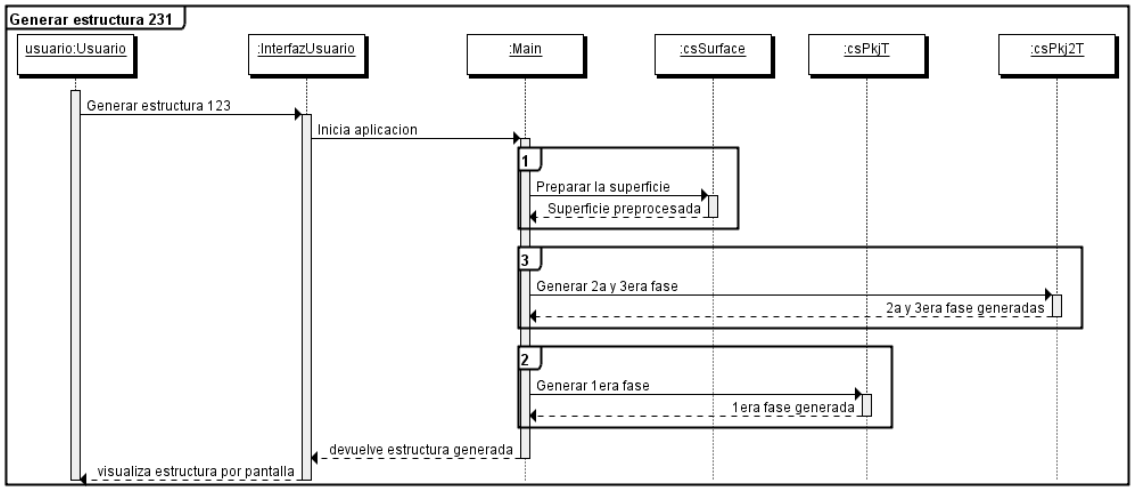

Figura 24. Diagrama de secuencia de "Generar estructura 231"

Este es exactamente igual que el anterior salvo que primero los puntos de unión entre la fase 1 y las fases 2 y 3 los elegimos nosotros.

Preparar estructura:

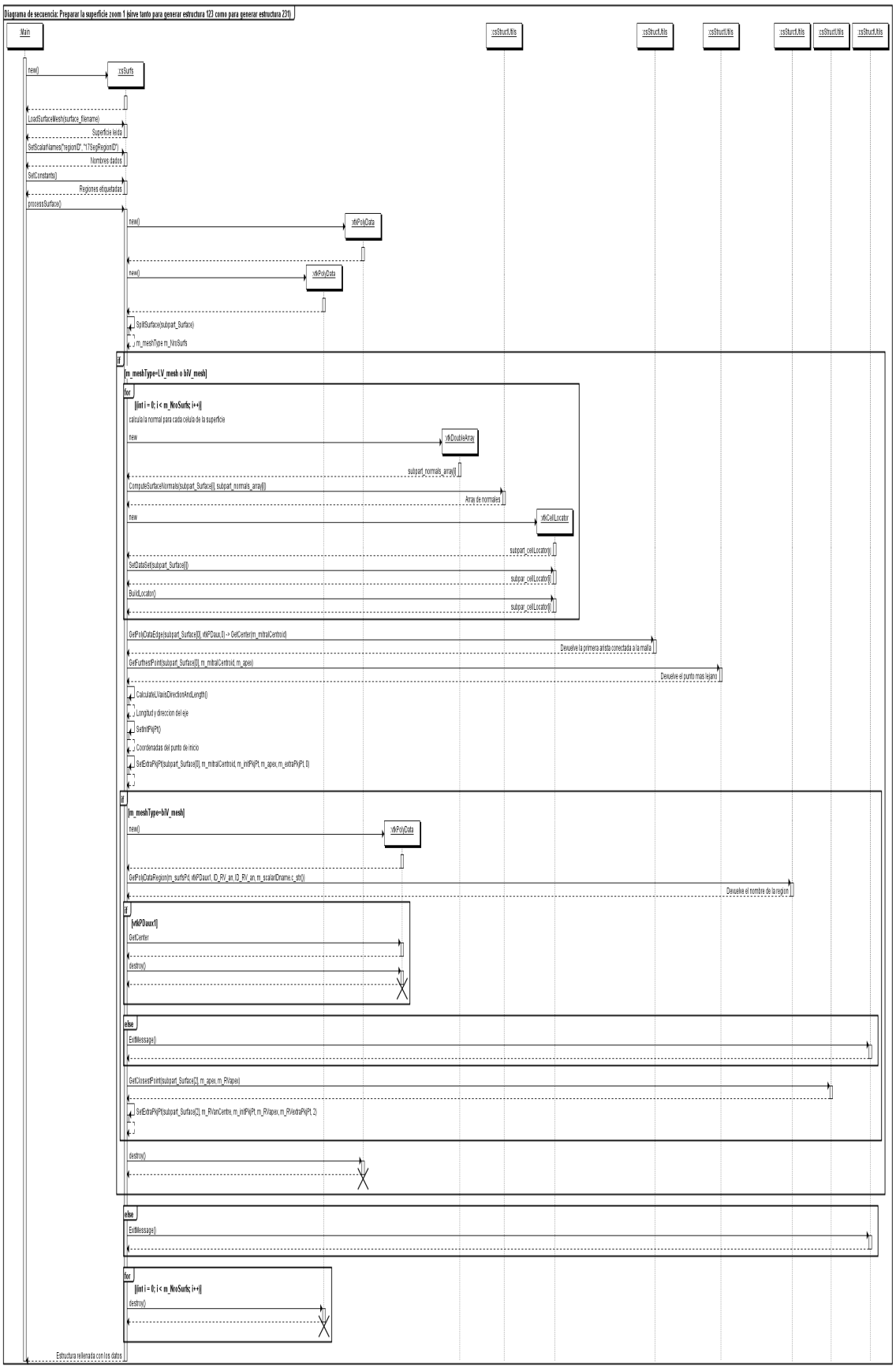

Figura 25. Diagrama de secuencia de "Prepara superficie"

Este diagrama ya se complica un poco puesto que ya intervienen objetos de tipo VTK así como llamadas a las clases de apoyo. Hay que constatar que cada llamada a csStructUtils representa un objeto nuevo creado.

Seleccionar puntos malla:

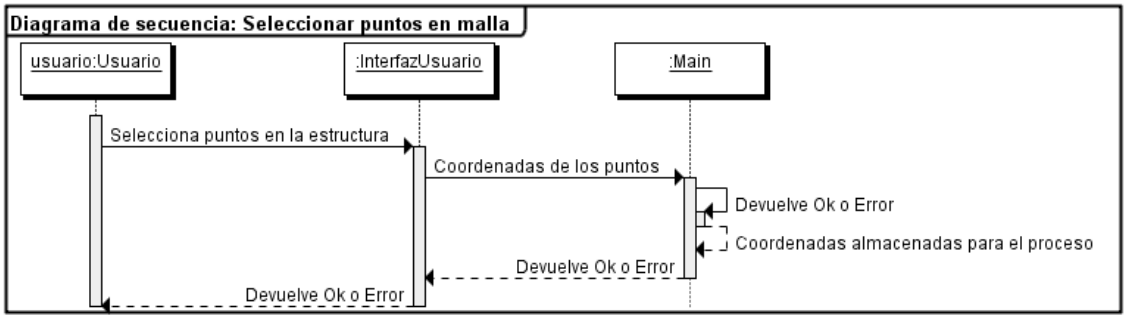

Figura 26. Diagrama de secuencia de "Seleccionar puntos malla"

Se nos permite seleccionar unos puntos (concretamente 3) de la imagen de la estructura vista en pantalla y almacenarlos para un posterior uso durante la generación de la fase 1.

Exportar estructura:

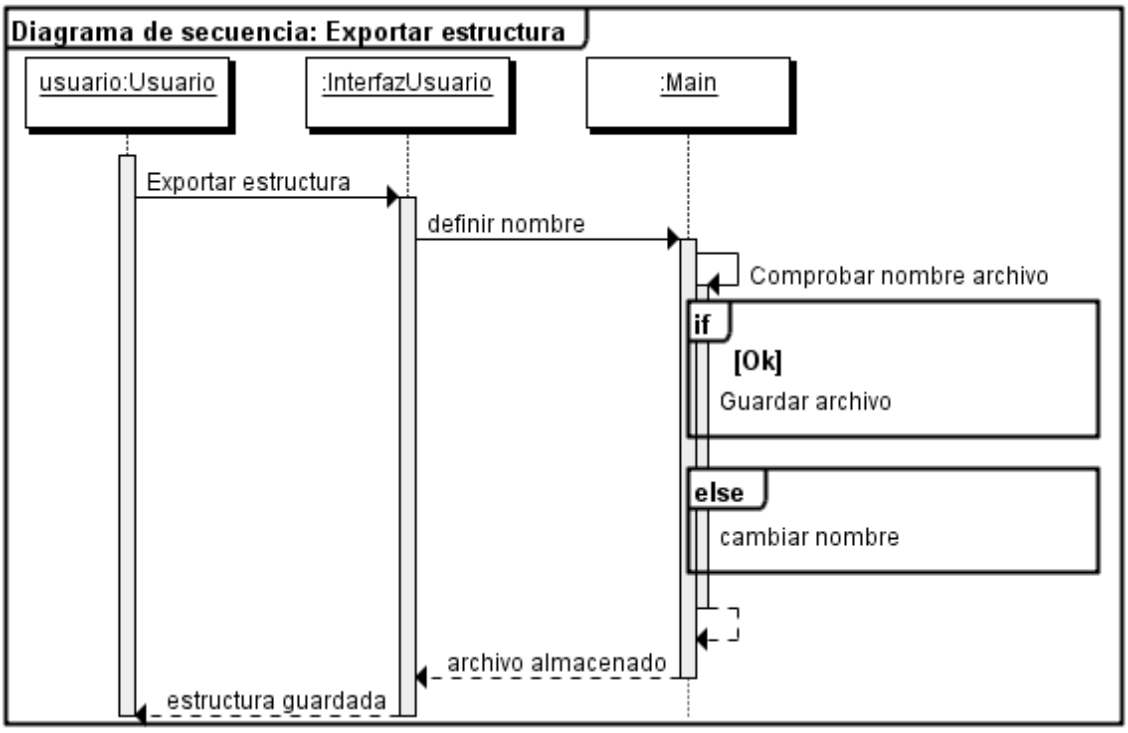

Figura 27. Diagrama de secuencia de "Exportar estructura"

Es la típica opción de guardar. Nos permitirá cargar la estructura generada en otro módulo para su posterior estudio o comparación.

Importar estructura:

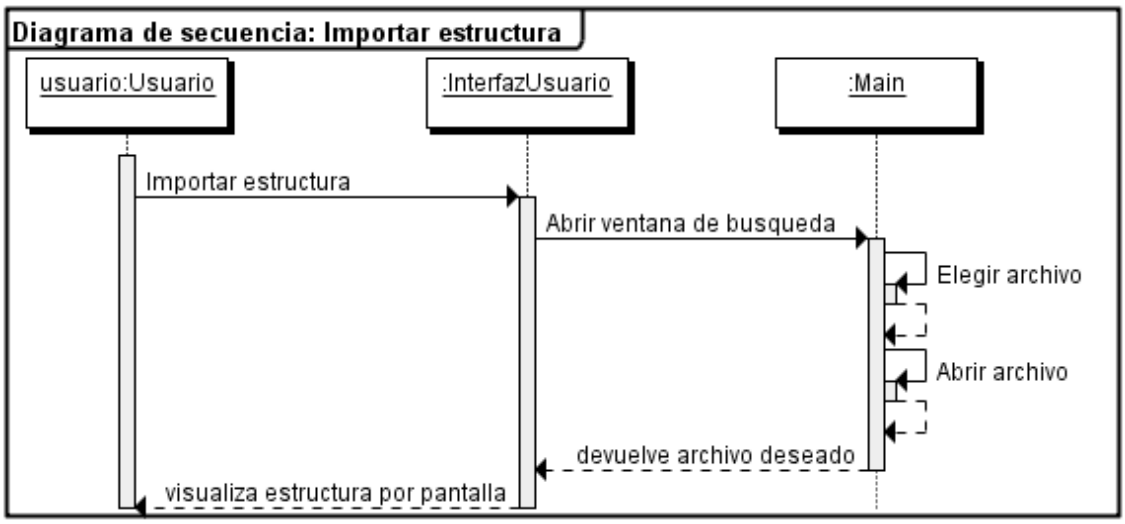

Figura 28. Diagrama de secuencia de "Importar estructura"

Típica función que nos permite cargar una estructura que hayamos almacenado con anterioridad.

Mostrar estadísticas:

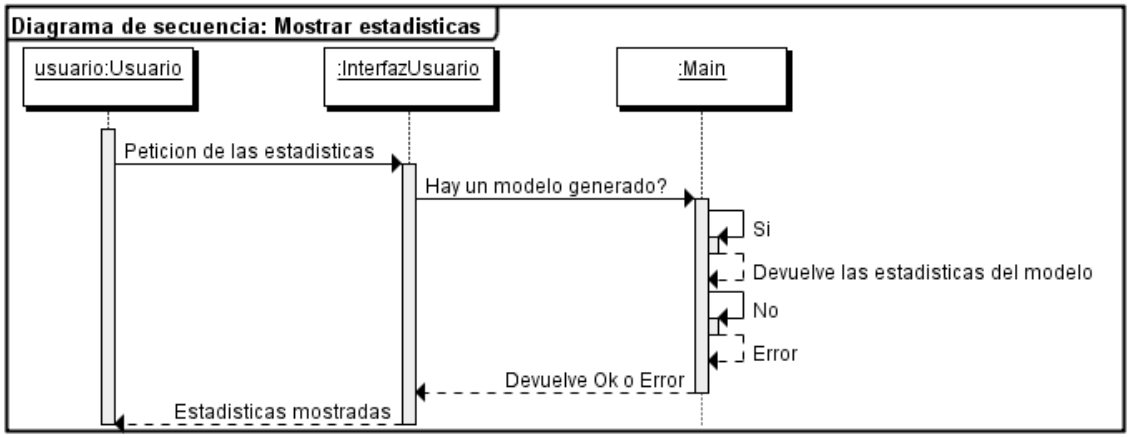

Figura 29. Diagrama de secuencia de "Mostrar estadísticas"

Muestra las estadísticas que se han calculado sobre el modelo que tenemos actualmente generado. Si no hay modelo no muestra las estadísticas.

5.2- Diseño

En este apartado, vamos a exponer, mediante diferentes tablas, los distintos casos de uso mediante tablas que nos darán más información. Posteriormente describiremos las ideas que tenemos para componer la interfaz gráfica.

5.2.1- Detalles de los casos de uso

Caso de uso Preparar estructura:

<table>
<tr><th>CU1-EM</th><th colspan="3">Preparar estructura</th></tr>
<tr><td>Descripción</td><td colspan="3">El sistema procesará la estructura inicial para, posteriormente utilizarla para generar nuestra red de Purkinje</td></tr>
<tr><td>Precondición</td><td colspan="3">Inicio de la generación = cierto</td></tr>
<tr><td rowspan="8">Secuencia Normal</td><td>Paso</td><td colspan="2">Acción</td></tr>
<tr><td>1</td><td colspan="2">Abrir directorio que contiene la malla</td></tr>
<tr><td>2</td><td colspan="2">Selecciona la malla que se desea cargar</td></tr>
<tr><td>3</td><td colspan="2">¿Existe la malla?</td></tr>
<tr><td rowspan="2"></td><td>3a</td><td>Sí. Devuelve ok</td></tr>
<tr><td>3b</td><td>No. Devuelve error</td></tr>
<tr><td>4</td><td colspan="2">Procesamos la respuesta obtenida de la pregunta anterior</td></tr>
<tr><td>5</td><td colspan="2">Realizamos la acción pertinente. Cargamos la malla seleccionada si existe y si no muestra un mensaje de error</td></tr>
<tr><td>Postcondición</td><td colspan="3">La malla existe o se genera un error</td></tr>
<tr><td rowspan="2">Excepciones</td><td>Paso</td><td colspan="2">Acción</td></tr>
<tr><td>3</td><td colspan="2">En el caso de que devuelva error, se tiene que permitir al usuario el elegir un archivo de malla distinto</td></tr>
<tr><td>Rendimiento</td><td colspan="3">El sistema deberá realizar la acción descrita lo más rápido posible puesto que es el primer paso necesario para la generación de nuestro modelo</td></tr>
<tr><td>Importancia</td><td colspan="3">Es uno de los pasos más importante de toda la aplicación</td></tr>
<tr><td>Urgencia</td><td colspan="3">Se tiene que realizar inmediatamente</td></tr>
<tr><td>Comentarios</td><td colspan="3">Es uno de los casos de uso más importante del sistema por no decir el más puesto que se tiene que realizar antes de poder generar nuestro modelo que al fin y al cabo el propósito principal de la aplicación</td></tr>
</table>

Tabla 7. Caso de uso de "Preparar estructura"

Caso de uso Generar estructura original:

CU2-GEO	Generar estructura original	
Descripción	El sistema generara la estructura de malla que queramos de forma ordenada. Es decir en primer lugar la fase 1, en segundo lugar la fase 2 y terminara con la fase 3	
Precondición	Malla previamente seleccionada	
Secuencia Normal	Paso	Acción
	1	Comprobamos que hemos seleccionado la malla correcta
	2	Modificamos si queremos el archivo de texto con las variables de entrada del sistema
	3	Leemos el archivo con los datos de los parámetros de entrada del sistema
	4	Pulsamos el botón correspondiente a generar estructura 123
	5	Generamos las tres ramas principales de la fase 1
	6	Generamos las ramas segundarias siguiendo la distribución de tipo *L-system*
	7	Completamos la malla hasta los limites introducidos por los parámetros de entrada o hasta que no quede más sitio en la malla
	8	Visualizamos la estructura por pantalla
Postcondición	Estructura generada correctamente	
Excepciones	Paso	Acción
	3	Al ser un software de investigación, no podemos certificar al 100% que los parámetros de entrada tengan valores correctos. Solo podemos controlar si no hay errores de tipos de datos
Rendimiento	El sistema debe generar la estructura de forma rápida y sin errores para que el usuario no se ofusque durante la espera	
Frecuencia	Este caso de uso se espera que se lleve a cabo siempre que el usuario lo desee. Esta totalmente bajo el control del usuario	
Importancia	Es muy importante puesto que el resultado servirá para posteriores estudios científicos	
Urgencia	No tiene mucha urgencia puesto que se realiza siempre que el usuario lo desee	
Comentarios	Este es uno de los casos de uso principales y para los que se ha constituido esta aplicación	

Tabla 8. Caso de uso "Generar estructura original"

Caso de uso Seleccionar puntos específicos:

CU-SPE	Seleccionar puntos específicos		
Descripción	Seleccionaremos tres puntos para su posterior uso en el siguiente caso de uso		
Precondición	Malla seleccionada y cargada correctamente		
Secuencia Normal	Paso	Acción	
	1	Cargamos la malla en la que queramos seleccionar los puntos específicos	
	2	Con el ratón del PC seleccionaremos tres puntos específicos siempre dentro del mismo cuadrante y sin sobrepasar una cierta altura del ventrículo	
	3	3a	Si los hemos seleccionado fuera del mismo cuadrante dará un error y permitirá volver a seleccionar los puntos
		3b	Si los puntos sobrepasan una cierta altura saltara un error y nos permitirá volver a seleccionarlos otra vez
	4	Almacenamos los puntos en variables	
Postcondición	Puntos correctamente seleccionados y almacenados		
Excepciones	Paso	Acción	
	2	En el caso de que nos salte algún error de selección, el sistema nos tiene que permitir volver a seleccionar otros puntos diferentes	
Rendimiento	La selección de puntos tiene que ser rápida una vez que el usuario lo haya hecho. Aunque la velocidad siempre dependerá de él		
Frecuencia	Tiene lugar como paso previo siempre que se vaya a generar una estructura alternativa		
Importancia	Es muy importante porque de él depende que podamos realizar el segundo caso de uso principal de la aplicación		
Comentarios	Digamos que este es un caso de uso transitorio puesto que tenemos que pasar por él para ir al caso de uso de generar estructura alternativa		

Tabla 9. Caso de uso "Seleccionar puntos específicos"

Caso de uso Generar estructura alternativa:

CU-GED	Generar estructura alternativa	
Descripción	El sistema generara la estructura de malla que queramos según los puntos que hemos elegido. Partiremos de los puntos seleccionados en el caso de uso anterior. Estos puntos serán la unión entre las diferentes fases	
Precondición	Puntos seleccionados correctamente	
Secuencia Normal	Paso	Acción
	1	Leemos las variables que contienen las coordenadas de los puntos previamente seleccionados
	2	Modificamos si lo deseamos los parámetros de entrada del archivo de texto
	3	Leemos los parámetros de entrada del fichero de texto
	4	Pulsamos el botón que lanzara el inicio de la creación de la estructura
	5	Partiendo de las coordenadas de los tres puntos previamente leídas, generamos la fase 1
	6	Generamos la fase 2 de la estructura
	7	Generamos la fase 3 de la estructura
	8	Visualizamos la estructura por pantalla
Postcondición	Estructura generada correctamente	
Excepciones	Paso	Acción
	2	Puede haber un error en la introducción de los caracteres de los parámetros de entrada. Saca un error e indica que se ha cometido en el fichero
	3	No se ha podido leer el fichero que contiene las coordenadas de los puntos. Sacamos un error e indicamos que no se ha podido leer correctamente
Rendimiento	El sistema debe generar la estructura de forma rápida y sin errores para que el usuario no se ofusque durante la espera	
Frecuencia	Este caso de uso se espera que se lleve a cabo siempre que el usuario lo desee. Esta totalmente bajo el control del usuario	
Importancia	Es muy importante puesto que el resultado servirá para posteriores estudios científicos. Sirve para probar otra metodología de creación de la malla	
Urgencia	No tiene mucha urgencia puesto que se realiza siempre que el usuario lo desee	
Comentarios	Este es el segundo de los casos de uso principales y para los que se ha constituido esta aplicación	

Tabla 10. Caso de uso "Generar estructura alternativa"

Caso de uso Importar estructura:

CU-IE	**Importar estructura.**	
Descripción	La aplicación carga el fichero contenedor de una malla que se ha generado previamente	
Precondición	El fichero vtk existe	
Secuencia Normal	Paso	Acción
	1	Pulsamos el botón que iniciará la búsqueda
	2	Se nos abrirá una ventana por la que navegar para buscar nuestro archivo
	3	Seleccionamos en archivo deseado una vez encontrado
	4	Pulsamos el botón abrir
	5	Se carga el fichero vtk además del fichero de estadísticas asociado al archivo
Postcondición	Fichero cargado correctamente	
Rendimiento	El sistema debe cargar la estructura leída del fichero de forma rápida	
Frecuencia	La frecuencia de uso de este caso es la que el usuario desea puesto que es él el que decide cuando cargar una estructura generada con anterioridad	
Importancia	Es relativamente importante puesto que si estamos realizando una investigación puntual de una malla concreta, no tendremos que generarla desde cero cada vez que queramos cargarla	
Comentarios	Es el clásico caso de uso de cargar un archivo ya existente. No tiene demasiada complicación puesto que el usuario medio debe de estar más que acostumbrado a su uso en multitud de aplicaciones	

Tabla 11. Caso de uso "Importar estructura"

Caso de uso Exportar Estructura:

CU-EE	Exportar estructura.	
Descripción	La aplicación almacena en un fichero vtk la estructura que hayamos creado en ese momento para de esta forma poder almacenarla y así poder usarla en otro momento	
Precondición	Estructura previamente generada correctamente	
Secuencia Normal	Paso	Acción
	1	Con la estructura generada correctamente por la aplicación, activamos el botón que iniciará el caso de uso
	2	Se nos abre una ventana para decidir donde guardamos le fichero
	3	Pulsamos el botón de guardar
	4	Guardamos el archivo con la estructura con una extensión vtk
	5	Guardamos en otro archivo con el mismo nombre todas las estadísticas
Postcondición	Fichero guardado correctamente	
Rendimiento	El sistema debe realizar el proceso de guardado de forma rápida y transparente. El usuario sólo se ha de preocupar del nombre que le pone al archivo	
Frecuencia	Este caso de uso se realiza cuando el usuario lo decida. Si no le interesa guardar la estructura, no lo realizará	
Importancia	No es muy importante puesto que se guarda sólo cuando el usuario quiera	
Comentarios	Es el clásico caso de uso de guardar un archivo cuando el usuario lo requiera. No tiene demasiada complicación puesto que el usuario medio debe de estar más que acostumbrado a su uso en multitud de aplicaciones	

Tabla 12. Caso de uso "Exportar estructura"

Caso de uso Ver estadísticas del modelo:

<table>
<tr><th>CU-VEM</th><th colspan="3">Ver estadísticas del modelo.</th></tr>
<tr><td>Descripción</td><td colspan="3">La aplicación nos mostrara toda la información referente a la estructura que hayamos creado</td></tr>
<tr><td>Precondición</td><td colspan="3">Existe la estructura a la que se refieren las estadísticas</td></tr>
<tr><td rowspan="6">Secuencia Normal</td><td>Paso</td><td colspan="2">Acción</td></tr>
<tr><td>1</td><td colspan="2">Tenemos una estructura generada y cargada correctamente</td></tr>
<tr><td>2</td><td colspan="2">El sistema busca de forma automática el fichero contenedor de la información</td></tr>
<tr><td rowspan="2">3</td><td>3a</td><td>Si el sistema encuentra el fichero, continua con la ejecución del caso de uso</td></tr>
<tr><td>2b</td><td>Si el sistema no encuentra el fichero, genera un error y vuelve a la pantalla principal</td></tr>
<tr><td>4</td><td colspan="2">Visualizamos por pantalla toda la información contenida en el fichero</td></tr>
<tr><td>Postcondición</td><td colspan="3">Información visualizada correctamente</td></tr>
<tr><td rowspan="2">Excepciones</td><td>Paso</td><td colspan="2">Acción</td></tr>
<tr><td>3</td><td colspan="2">En el caso de que no se encuentre el archivo deseado, generar una excepción que será la encargada de tratar el error y gestionarlo</td></tr>
<tr><td>Rendimiento</td><td colspan="3">El sistema cargará la información de forma rápida para poder tenerla al lado del modelo 3D de la estructura para de esta manera poder maximizar el estudio de investigación</td></tr>
<tr><td>Frecuencia</td><td colspan="3">Se realiza siempre que generemos una estructura por lo que la frecuencia de uso dependerá de del usuario</td></tr>
<tr><td>Importancia</td><td colspan="3">Es muy importante puesto que gracias a esta información, podremos tener un mejor control de nuestra estructura viendo toda la información necesaria</td></tr>
<tr><td>Urgencia</td><td colspan="3">Se realiza inmediatamente después de terminar de generar la estructura</td></tr>
<tr><td>Comentarios</td><td colspan="3">Es un caso de uso muy relevante puesto que gracias a él podemos visualizar al mismo tiempo la estructura y toda su información para tener un mayor control de la situación</td></tr>
</table>

Tabla 13. Caso de uso "Ver estadísticas del modelo"

Caso de uso Interactuar con el modelo:

CU-IM	Interactuar con el modelo.	
Descripción	El sistema permite al usuario el interactuar con el modelo 3D que haya generado. Permite rotar y hacer zoom sobre él mediante el uso del ratón	
Precondición	Estructura en cargada en pantalla	
Secuencia Normal	Paso	Acción
	1	Tenemos una estructura cargada en el interfaz gráfico
	2	Comenzamos a interactuar con ella gracias al ratón
	3	Se recarga la estructura una vez terminada la interactuación
Postcondición	Posición o zoom de la figura modificado	
Rendimiento	El sistema tiene que responder al instante. No puede haber tirones o desfases muy destacables	
Frecuencia	El usuario no siempre interactuará con la figura	
Importancia	Tiene una importancia relevante puesto que no tiene porque ser necesario rotar o acercar la cámara para ver el modelo. Más bien con este caso de uso conseguiremos apreciar más los detalles	
Comentarios	Este caso de uso describe las diferentes interactuaciones que vamos a ser capaces de realizar una vez que tengamos una estructura generada y cargada en pantalla para su visualización. Solo tenemos dos maneras diferentes de interactuar con la figura por lo que no creemos que resulte muy difícil de aprender. El código que controla este caso de uso esta implementado en la aplicación global de GIMIAS por lo que no tendremos ningún diagrama de secuencias al respecto	

Tabla 14. Caso de uso "Interactuar con el modelo"

5.2.2- Diseño de la interfaz gráfica

A la hora de interactuar con nuestro *plugin*, disponemos de diferentes *widgets* que nos permitirán realizar las acciones asociadas a los diferentes casos de uso anteriormente descritos. Al ser una aplicación *OpenSource*, dichos *widgets* pueden ser creados desde cero por los desarrolladores o modificados de los que ya existen. Para ello, GIMIAS provee un API para los *plugins* para que interactúen con el núcleo de la aplicación y/o con otros *plugins* del *framework*.

Estos *plugins*, necesitan tres clases diferentes para crearlos: la clase *Widget* (la cual provee un elemento de la interfaz gráfica del usuario, ordenando el panel acorde a las necesidades), la clase *FrontEndPlugin* (la cual provee las

funcionalidades principales del *plugin*) y la clase Processor (que ejecuta un algoritmo con una o múltiples entradas y genera una o múltiples salidas).

Cada *plugin* se crea como una clase derivada de la clase *FrontEndPlugin*. Contienen una o varias clases de tipo *Widget* (dependiendo de los que hayamos creado), y una o varias clases de tipo Processor. Los datos se almacenan y manejan usando las siguientes clases: DataEntity (almacena y encapsula un solo objeto de datos), DataHolder (permite a los objetos ser registrados como observadores de datos para notificar cuando varían los datos) y DataEntityList (contiene una lista con los datos de la entidades actualmente permitidas para el proceso en la aplicación. En la lista se almacenan tanto los datos de entrada como los de salida. Esta lista es accesible a todos los *plugins*). Para ver mejor como se realiza dicha interactuación vamos a ver una figura que representa el caso de uso del proceso de datos en GIMIAS (fig. 30)

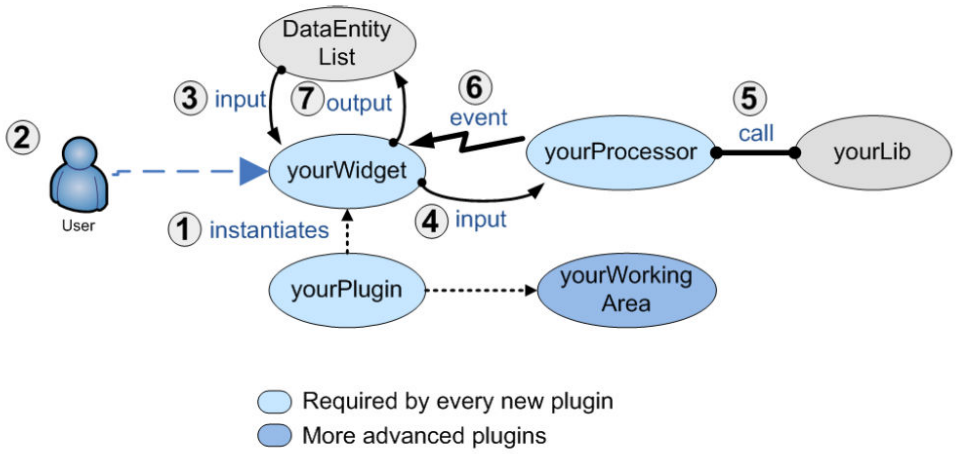

Figura 30. Caso de uso de proceso de datos en los *plugin*s de GIMIAS

En la siguiente figura (fig. 31) vemos un resumen de las clases que pueden ser usadas y/o extendidas para la creación de *plugins*.

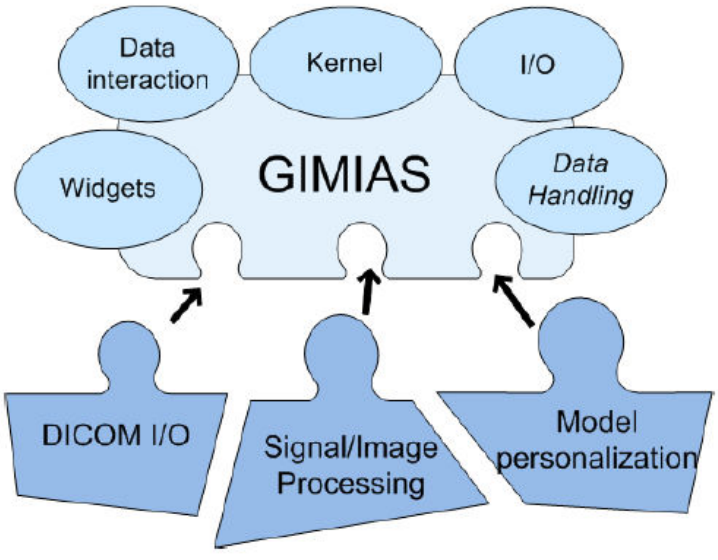

Figura 31. Componentes del *framework* de GIMIAS

Todos los *plugins* gráficos de GIMIAS tienen el mismo estilo por lo que nosotros vamos a seguirlo pues consideramos que si los usuarios ya están familiarizados con un cierto estilo mejor no variarlo para su comodidad.

El nuestro no va a ser menos puesto que queremos que predomine la usabilidad y sencillez de uso. Por ello seguiremos la misma estructura (fig. 32).

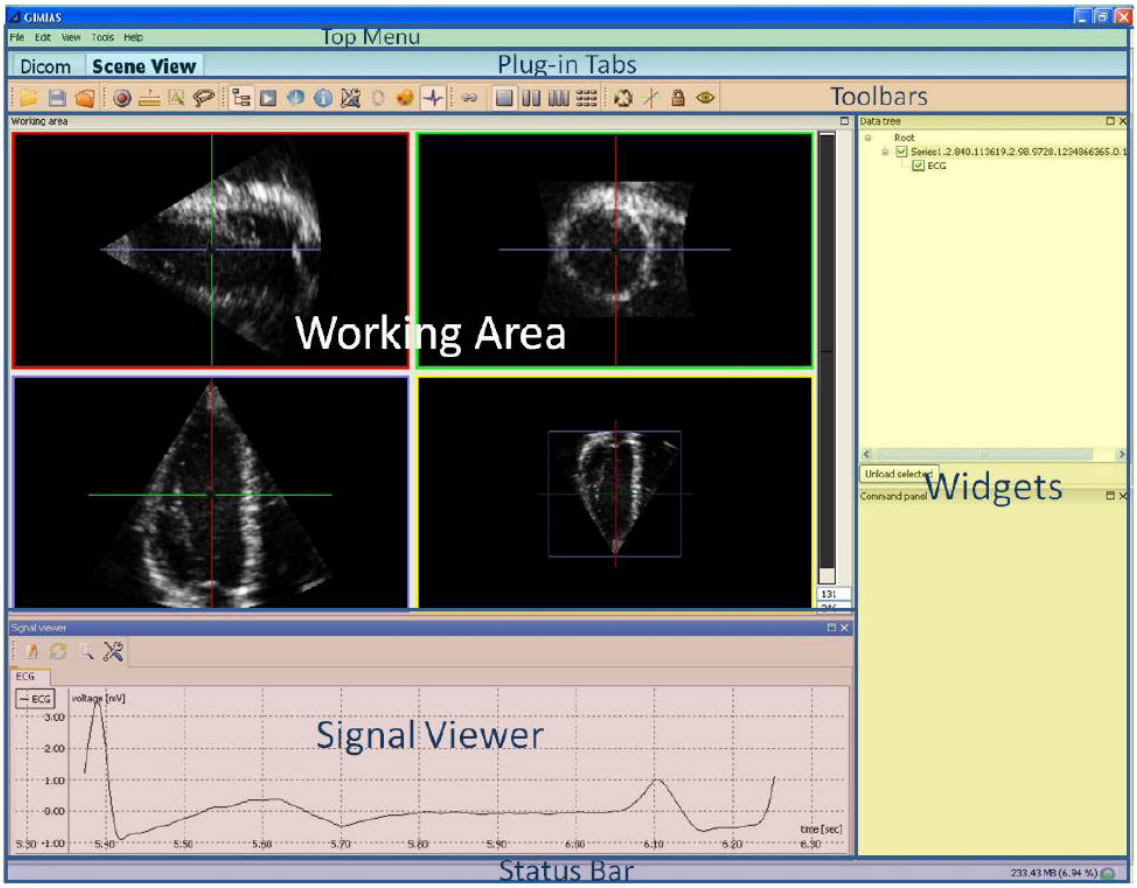

Figura 32. Interfaz gráfica estándar de GIMIAS

La finalidad del proyecto es la construcción y posterior visualización de una red de *Purkinje*. Por eso consideramos que la parte más importante de nuestra aplicación será el espacio de visualización.

Si observamos la figura anterior, detectamos que el área de trabajo está puesta a la izquierda de la interfaz gráfica y cubre sobre el 60%-75% de la pantalla. Esto se debe a la finalidad del software. Como hemos dicho una de las finalidades de GIMIAS es el estudio e interpretación de imágenes médicas. Por ello el espacio para dicha función tiene que ser notorio.

En la parte superior tenemos la barra de menús de GIMIAS y justo debajo la barra con las pestañas de los *plugins* que tenemos activos para poder navegar entre ellos. Más abajo, tenemos una barra de menús para interactuar con nuestro *plugin*.

En la mitad derecha inferior, estará nuestro menú, con nuestros *widgets*, para poder activar los diferentes casos de uso de nuestra aplicación. Se separaran mediante líneas horizontales y/o pestañas.

Vamos a partir de este modelo estándar para configurar nuestro propio *plugin*. De esta manera conseguiremos que sea muy accesible a los usuarios.

Todo esto lo haremos con una aplicación muy sencilla llamada wxGlade que ya está preparada para crear los menús de los *widgets* de forma gráfica.

Posteriormente implementaremos el código para poder enlazar nuestras clases a los botones creados. De esta forma cuando pulsemos un botón determinado, lanzaremos los métodos necesarios. Todo esto lo veremos en el siguiente punto donde veremos con más detalle todo lo referente a la implementación de la aplicación.

5.3- Implementación

En este apartado vamos a explicar cómo ha sido la implementación de nuestro proyecto. En primer lugar explicaremos las clases, para después pasar al *plugin* y finalmente a la creación de la interfaz gráfica.

Lo primero que hay que hacer antes que nada es generar los ficheros de nuestro proyecto. Para ello usaremos una aplicación llamada “StartNewModule” (fig. 33). Gracias a esta pequeña aplicación se nos va a generar la estructura de carpetas común para todos los proyectos de GIMIAS vista anteriormente en la figura 17.

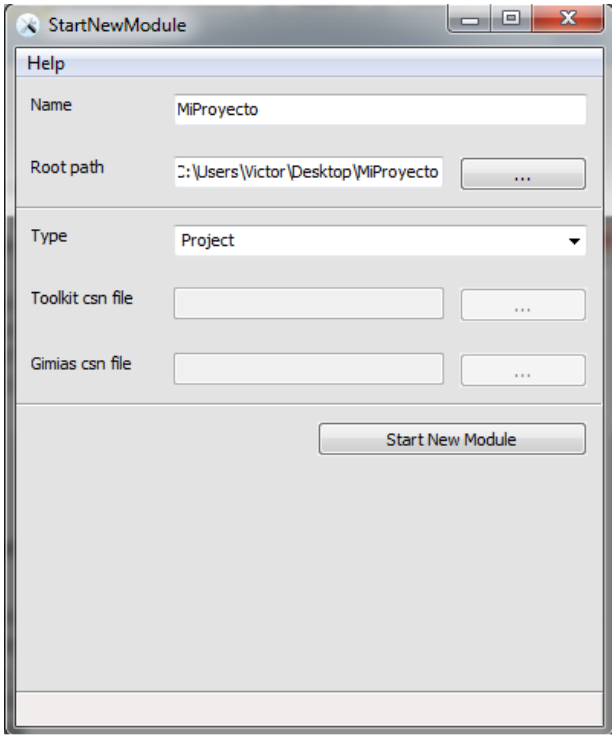

Figura 33. Creación de los archivos del proyecto

Una vez con la estructura de carpetas (fig. 34) ya creada, vamos a dar paso a la implementación.

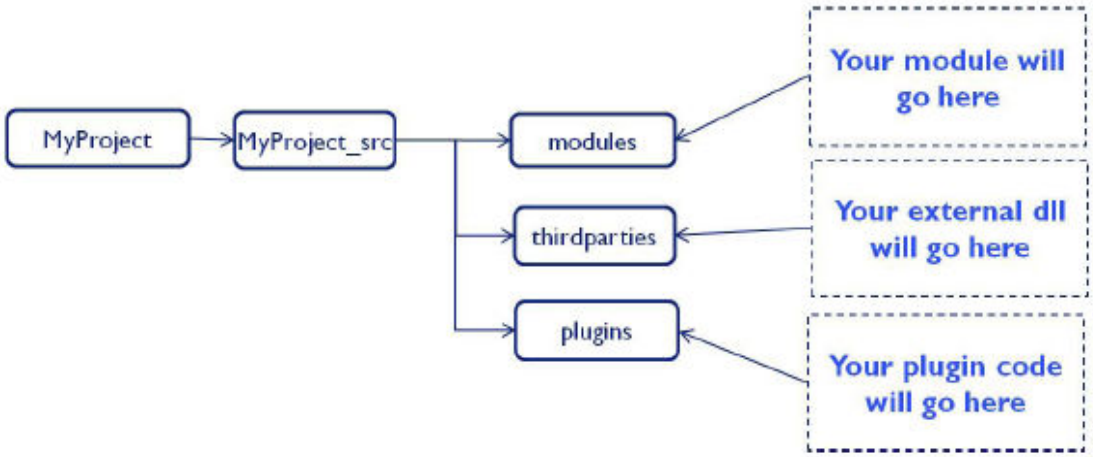

Figura 34. Estructura de las carpetas de nuestro proyecto.

5.3.1- Implementación de las clases

Como ya hemos comentado, nuestra idea es la de tener cinco clases principales para poder crear nuestra solución: csPkjT, csPkj2T, csPkjTime, csStructUtils y csSurfaces. Las dos primeras controlan las propiedades necesarias para la generación de la estructura. La tercera nos va a servir para controlar los tiempos de creación. La cuarta nos ofrece una serie de métodos que nos van a servir para realizar ciertas comprobaciones matemáticas. La última es muy necesaria puesto que nos preprocesara la superficie de la estructura para poder dividirla en regiones que posteriormente nos servirán para controlarla.

Para crearlas, vamos a utilizar una aplicación sencillísima llamada "StartNewModule" (fig. 35) que nos va a generar los archivos .h y .cxx con el código esencial que se va a necesitar. Usando estos ficheros como base de partida, iremos añadiendo nuestro código.

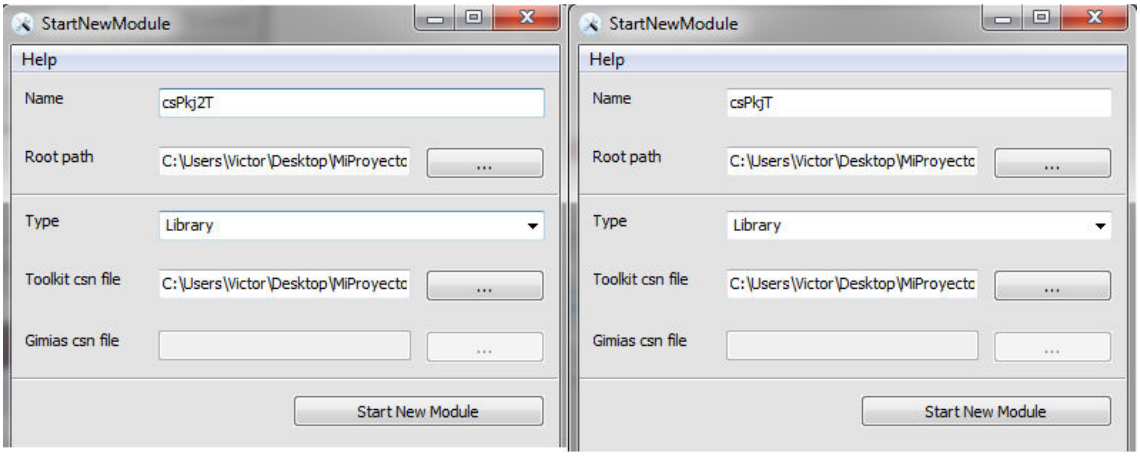

Figura 35. Creación de los archivos de las clases csPkjT y csPkjT2

Las clases se generarán en la carpeta llamada “modules” que hemos visto en la figura 31. Dentro de esta carpeta se nos van a generar otras subcarpetas que van a contener diferentes archivos (fig. 36).

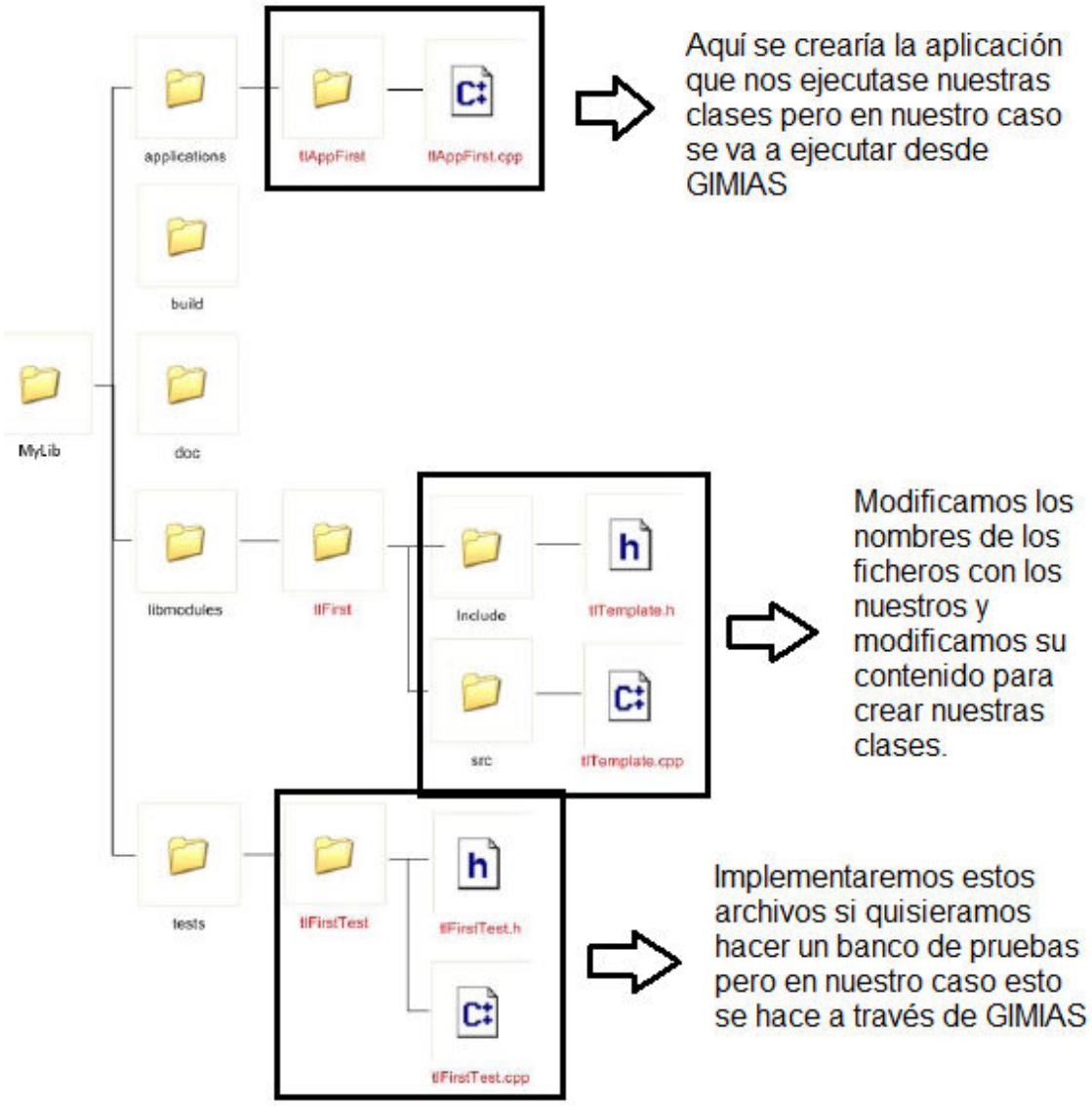

Figura 36. Estructura de carpetas de nuestras librerías

A continuación vamos a ir detallando cada una de ellas para explicar, dentro de su complejidad, los métodos que las componen.

Clase csPkjT:

Esta es la clase encargada de generar la primera fase de nuestra red de *Purkinje*. Contiene los métodos necesarios para realizar las comprobaciones pertinentes para crear las tres ramas principales de las cuales posteriormente colgarán las ramas generadas tanto en segunda como en tercera fase. Dichas ramas partirán desde las coordenadas correspondientes al punto del corazón conocido como: nodo atrioventricular (visto en la figura 1).Esta clase tiene como método principal a “BuildPkjTree”.

Contiene los atributos necesarios para el cálculo y creación de las ramas como puede ser la longitud, los valores de los ángulos que se usan para el crecimiento de las ramas con el *L-system*, vectores con los posibles puntos, etc… A continuación vamos a ver una lista con todos los atributos de la clase así como un breve comentario de cada uno.

Atributos de la clase csPkjT:

m_maxAngle: máximo ángulo de apertura para las ramas nuevas.
m_minAngle: mínimo ángulo de apertura para las ramas nuevas.
m_sigmaA: desviación estándar para el ángulo de la siguiente sub parte de una rama.
m_sigmaA0: desviación estándar del ángulo para la rama inicial.
m_divLength: longitud de cada división.
m_sigmaDiv: desviación estándar para el número de divisiones que giran hacia el mismo lado.
m_frctL0min: valor mínimo de L0 como una fracción de L0 de las ramas 1 y 2.
m_frctDist4loop: fracción de la distancia al cruce permitido de puntos. Controla los bucles de ramas.
m_minPctgL: porcentaje de m_divLength que determina la mínima distancia entre puntos permitidos.
m_gradFrctL: fracción de m_divLength cuando calculamos el campo gradiente.
m_turnW: valor del peso asignado al campo distancia.
m_maxAngCurve: máxima desviación del ángulo de cada sub parte de la rama con respecto a la sub parte previa.
m_maxAngTotCurve: máximo total de giros de ángulos para cada rama.
m_nrNearPts: máximo número de puntos cercanos permitidos.
m_minDist2base: mínima distancia a la base permitida.
m_sigmaD2base: desviación estándar para m_minDist2base.
m_maxBrLxZlength: fracción de Zlength para determinar la longitud máxima de las ramas.
m_muL_IGD: parámetros mu para la distribución Gausiana inversa de la longitud de las ramas.
m_lambdaL_IGD: parámetros lambda para la distribución Gausiana inversa de la longitud de las ramas.
m_xoL_IGD: parámetros x0 para la distribución Gausiana inversa de la longitud de las ramas.
m_muP_IGD: parámetros mu para la distribución Gausiana inversa.
m_lambdaP_IGD: parámetros lambda para la distribución Gausiana inversa.
m_xoP_IGD: parámetros x0 para la distribución Gausiana inversa.
m_muL0fr: parámetros denominadores mu para la posición L0 – distribución normal.
m_sigmaL0fr: parámetros denominadores sigma para la posición L0 – distribución normal.
m_muL0fr0: denominador mu para la L0 pos – n dist. Primeras ramas.
m_sigmaL0fr0: denominador sigma para la L0 pos – n dist. Primeras ramas.

m_PassPtsWeight: valor del peso para la redirección de las primeras ramas hacia los puntos de paso.

m_FrctNonPassPt: sección de la segunda y tercera rama no redirigidas hacia los puntos de paso.

m_nrDivsSameAngle: número de segmentos que giran hacia el mismo lado.

m_minNrDiv: número mínimo de divisiones para cada rama.

m_Nbranchs: número de ramas (aproximativo).

m_fstBrInBr1: primera rama en las ramas "1" y "2" después de dividirse en bucles para permitir a las ramas dirigirse directamente a la base en el septum.

m_lstBrInBr2: última rama en las ramas "1" y "2" después de dividirse en bucles para permitir a las ramas dirigirse directamente a la base en el septum.

m_verb: valor de control.

typedef struct BranchStructure: estructura contenedora de la información de las estructura de las ramas.

Como podemos ver, tenemos un número considerable de atributos de clase a tener en cuenta a la hora de crear nuestras ramas. Esto es debido al intento de representar la realidad biológica lo más precisa posible gracias al método *L-system*. Todos estos atributos, son usados en los métodos de la clase para ir generando la estructura de la red de *Purkinje*.

El método "**BuildPkjTree**" es el encargado de ir llamando a los demás métodos para ir generando las tres ramas principales de la estructura. Aquí iniciamos el contador de tiempo para posteriormente recuperar su valor y saber cuánto ha tardado. Se define una variable contador para saber cuántas ramas hemos creado Nos interesa diferenciar las tres ramas iniciales del resto.

Acompañando la descripción de los métodos, hay una serie de figuras representando las secuencias de ejecución para así intentar visualizarlo mejor.

Primero se decide qué dirección van a tomar las nuevas ramas mediante el método "**decidesDirection**". Dentro de esta función, se calcula la normal a los puntos de la superficie y, si el punto es RV, comprueba la etiqueta de la región. Posteriormente, dependiendo del valor del parámetro I que se le pasa al método, realizará unos cálculos u otros. Este valor I nos va a determinar en qué punto de la estructura estamos. Nos devuelve un vector de tipo csStructUtils con los valores de las direcciones de las ramas. Vamos a ver cómo sería esto en la siguiente figura (fig. 37).

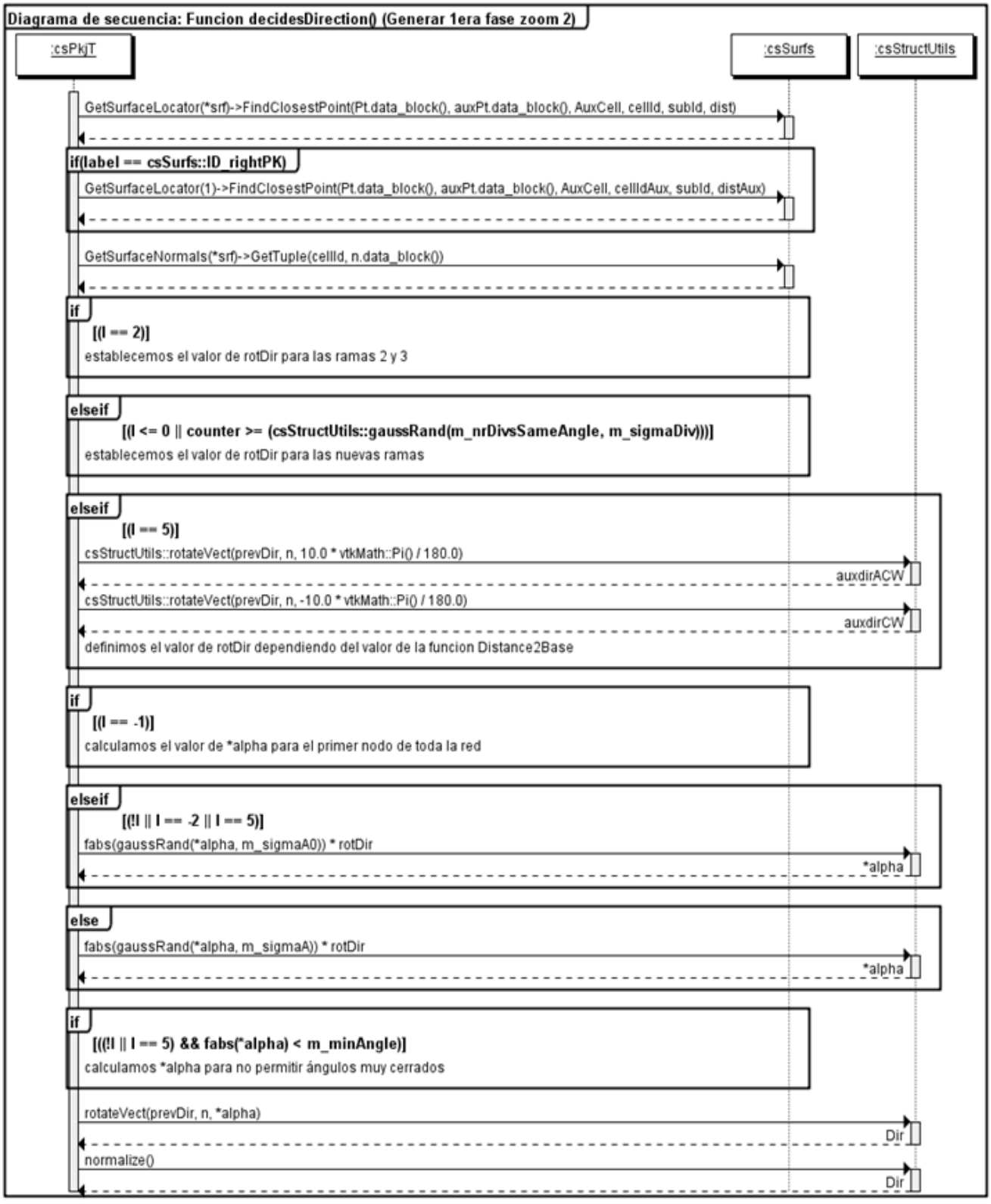

Figura 37. Diagrama de secuencia del método decidesDirection

El siguiente método que se ejecuta es "**decidesStartPt**" (fig. 38). Esta función nos calcula las coordenadas de los nuevos puntos de las ramas. Nos devuelve el valor de dichas coordenadas y la dirección previa por referencia y la longitud inicial de la rama

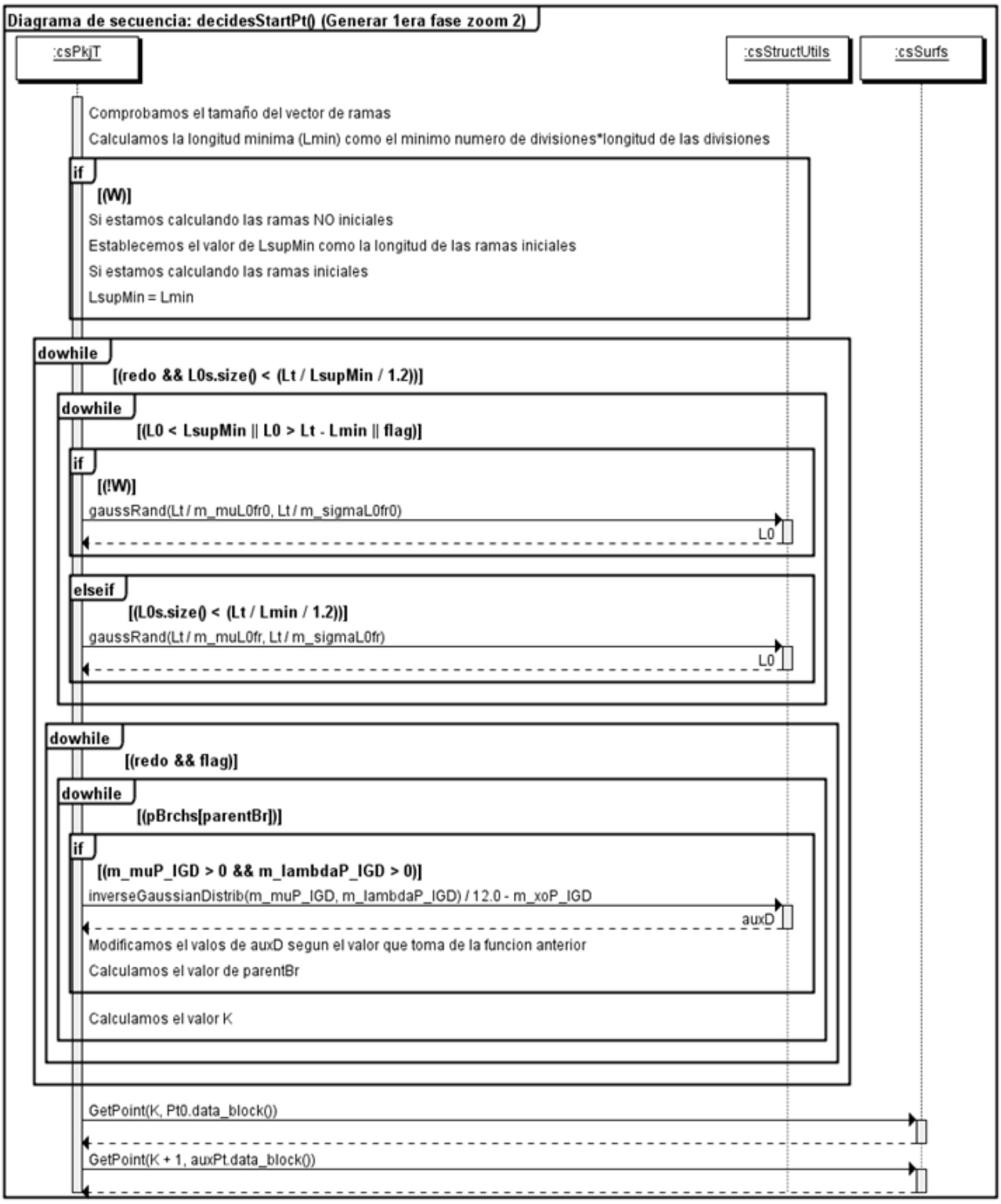

Figura 38. Diagrama de secuencia del método decidesStartPt

A continuación se ejecutará el método "**growBranch**" que hace que crezcan las ramas. Este método genera las tres ramas principales de la primera fase, uniendo el punto de inicio, que es común para las tres ramas, y el punto final correspondiente de cada una. Las ramas se van generando en diferentes direcciones (predominando el sentido descendente). Cada punto final está presente en regiones diferentes y, según los conocimientos médicos, nunca pueden sobrepasar una altura dada. Esto se realiza con la ayuda de otros métodos que explicare a continuación (fig. 39).

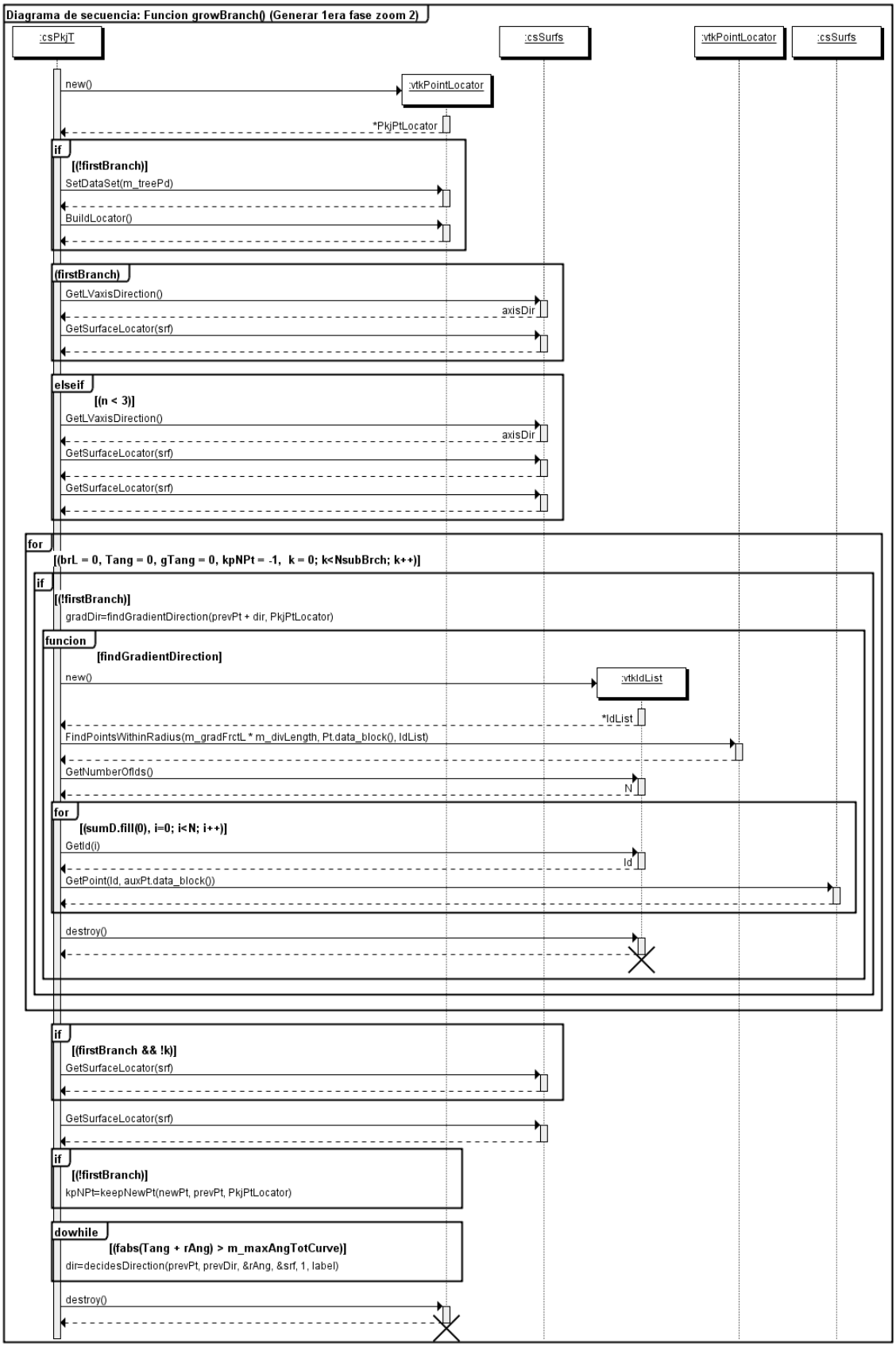

Figura 39. Diagrama de secuencia del método growBranch

El primer método que encontramos dentro de *growBranch* es "**findGradientDirection**". Este método nos sirve para encontrar los puntos situados dentro de un determinado círculo de radio determinado y centro un punto x determinado. Dichos puntos son recopilados mediante un identificador que tienen. Posteriormente, nos calcula la dirección principal del punto central a los que están a su alrededor. Nos devuelve un vector con las direcciones a esos puntos.

Otro método que nos ayuda a generar las ramas es "**keepNwPt**". Sirve para poder comprobar si los puntos nuevos que vamos encontrando nos sirven o no. Se calcula la distancia entre el punto previo por donde pasa la rama y la siguiente posibilidad. Si el punto está lo suficientemente lejos y no corta el árbol que existe, se mantiene y se sigue creciendo la rama. Si el nuevo punto está demasiado cerca de otro punto, se para de crecer desde ese punto. Si no nos interesa, se borra. Si se tiene que mantener pero genera un bucle, nos devuelve el identificador del punto donde se cierra el bucle (fig. 39).

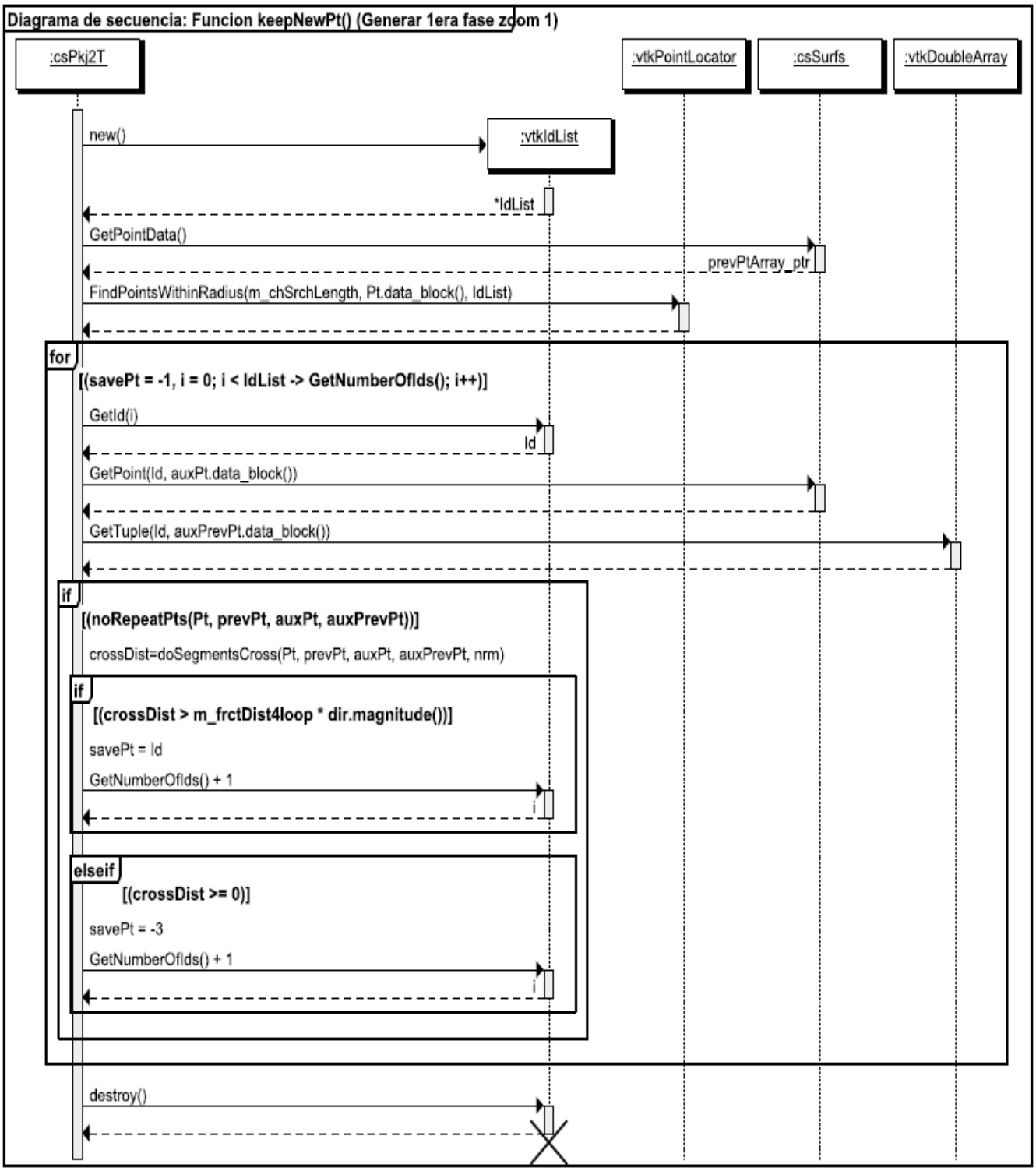

Figura 40. Diagrama de secuencia de keepNwPt

Para comprobar si dos segmentos se cortan, usamos el método "**doSegmentCross**". Nos devuelve un valor positivo si los dos segmentos (constituidos por dos puntos cada uno) no se cortan. Si en cambio si lo hacen devuelve un valor negativo.

Otro método para comprobar opciones sobre los puntos es "**noRepeatPts**". Nos sirve para realizar la sencilla verificación de si cuatro puntos dados se

repiten. Nos devuelve cierto si los cuatro puntos son diferentes entre ellos y falso en otro caso.

Para el tratamiento sobre las ramas tenemos "**divideParentsBrs**". Este método nos permite dividir una rama en varias cuando alcanzamos cierto punto. Dicho punto pasará a ser el final de la primera rama y el inicio de las nuevas. Posteriormente determina la primera y última rama de las previas (fig. 41).

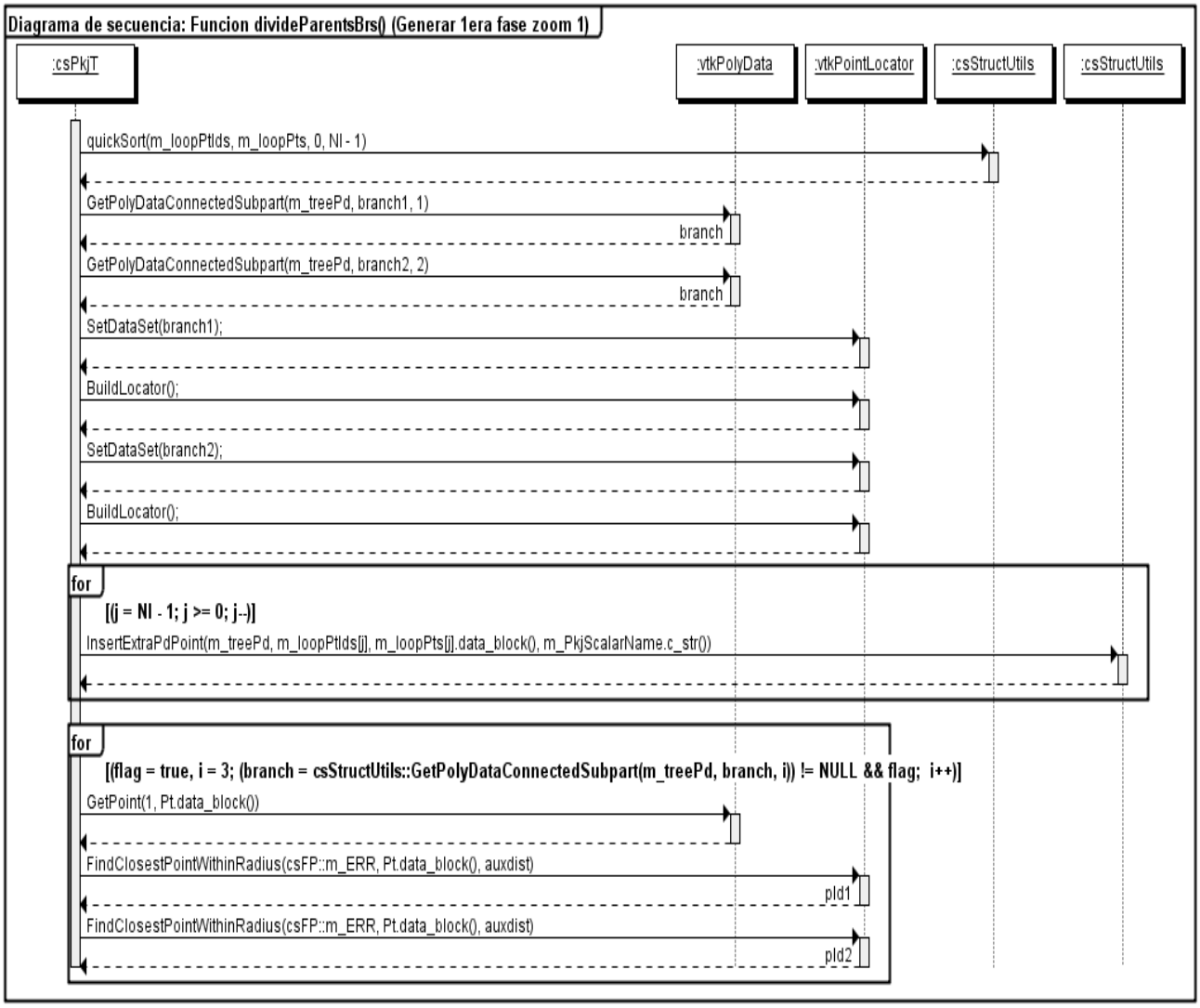

Figura 41. Diagrama de secuencia de divideParentsBrs

El último método que nos encontramos en esta clase es "**Distance2Base**". Esta función nos calcula la distancia entre un punto determinado a la base de la estructura (actualmente al plano definido por el centro de la mitral y perpendicular a los ejes principales.

Con esto concluimos los métodos presentes dentro de la clase csPkjT que nos permiten generar la primera fase de nuestra estructura.

Para generar la segunda y tercera fase, tenemos la clase csPkj2T.

Clase csPkj2T:

Esta clase tiene una mayor complejidad que la anterior. Más por el hecho que tiene que controlar un número superior de ramas mientras se crean por lo que tiene que estar pendiente de muchos más detalles. Además la clase anterior sólo tiene que genera una fase pero esta tiene que generar dos. La segunda genera un número determinado de ramas (que le hemos pasado por parámetros) y partiendo de los puntos que constituyen las ramas de la primera fase (pero siempre teniendo en cuenta que las nuevas ramas nunca podrán sobrepasar una altura determinada). La tercera fase rellena aun más los espacios que puedan haber quedado (controlando los bucles y siempre que se puedan generar ramas). La generación finalizará cuando se hayan cumplido los parámetros de entrada o cuando la estructura no acepte más ramas nuevas. Todo esto se realiza mediante una serie de atributos y métodos.

Atributos de la clase csPkj2T:

m_angle: ángulo de apertura de las nuevas ramas.
m_sigmaA: desviación estándar del ángulo.
m_1stLangle: ángulo de apertura de las ramas para la primera línea como segundo paso después de csPkjT.
m_1stLsigmaA: derivación estándar para el ángulo m_1stLangle.
m_length: longitud de la rama.
m_sigmaL: desviación estándar para la longitud.
m_minPctgL: porcentaje de m_length que determina la distancia mínima entre puntos permitidos.
m_gradPctgL: porcentaje de m_length para calcular el campo gradiente.
m_PctgL4loops: porcentaje de una longitud para puntos cercanos que todavía no se cruzan formando bucles.
m_turnW: valor del peso asignado al campo de distancia de giro.
m_maxBrLength:
m_MaxAngleDeviation: máxima desviación del ángulo de cada sub parte de la rama con respecto a la sub parte previa.
m_nrNearBranches: número de ramas próximas a un punto.
m_minDist2base: distancia mínima a la base.
m_sigmaD2base: desviación estándar para la distancia mínima a la base.
m_brLseptum: longitud de la rama limitada para el septum
m_frctDist4loop: fracción de la distancia al cruce permitido de puntos. Controla los bucles de ramas.
m_chSrchLength: longitud buscada para puntos que podrían cruzar segmentos.
m_minDist2initPt: distancia mínima a puntos iniciales permitidos para nuevos terminales.
m_nrSubBrnch: número de segmentos por rama.
m_growRule: regla de crecimiento.
m_nBrchs: número de bucles en “newBranches”.
m_nroNewTerms: número de nuevos terminales que se van a crear después de crear las ramas principales (para la segunda fase).
m_fstSpecTerm, m_lstSpecTerm: primeros y últimos nuevos terminales en las ramas “1” y “2” para evitar que las ramas se dirijan a la base en el septum.

m_mxNmbrNearPts: número máximo de puntos cercanos permitidos cuando creamos nuevos terminales (junto con m_chSrchLength).
m_verb: valor de control.
m_first: punto de control
m_ApplyRule, m_BranchDir: reglas y dirección de las ramas
m_brTLength: vector con la longitude total de cada rama.
m_newPts[2]: puntos previos para dibujar las líneas.
m_loopPtIds: identificadores de los puntos de bucle
m_loopPts, m_nonGrewPts: puntos de bucle y de no crecimiento
m_PkjScalarName: nombre del escalar
m_branch1, m_branch2: primera y segunda rama

Al igual que en la clase anterior, tenemos un gran número de atributos para poder representar de la manera más fiel posible las características fisiológicas del corazón y de esta manera intentar obtener un software lo más cercano a la realidad.

Para generar las ramas disponemos de una serie de métodos que nos realizaran las operaciones pertinentes. El primero que se ejecuta es "**Build2ndStepOfPkjTree**". Como su nombre indica es el encargado de generar el segundo paso de la estructura (fases dos y tres). A partir de esta función se irán ejecutando el resto hasta completar nuestro corazón en 3D. Todos los métodos que vamos a exponer a continuación se usan tanto en la segunda y la tercera fase de la generación. Sólo hay uno que se difiere y lo explicaremos al final.

Para iniciar la generación de las ramas de la segunda fase, se lanza "**createNewTerminals**". Se crean nuevos terminales en las ramas. Es un método muy importante puesto que va a generar los nuevos puntos de unión de nuestro modelo (fig. 42)

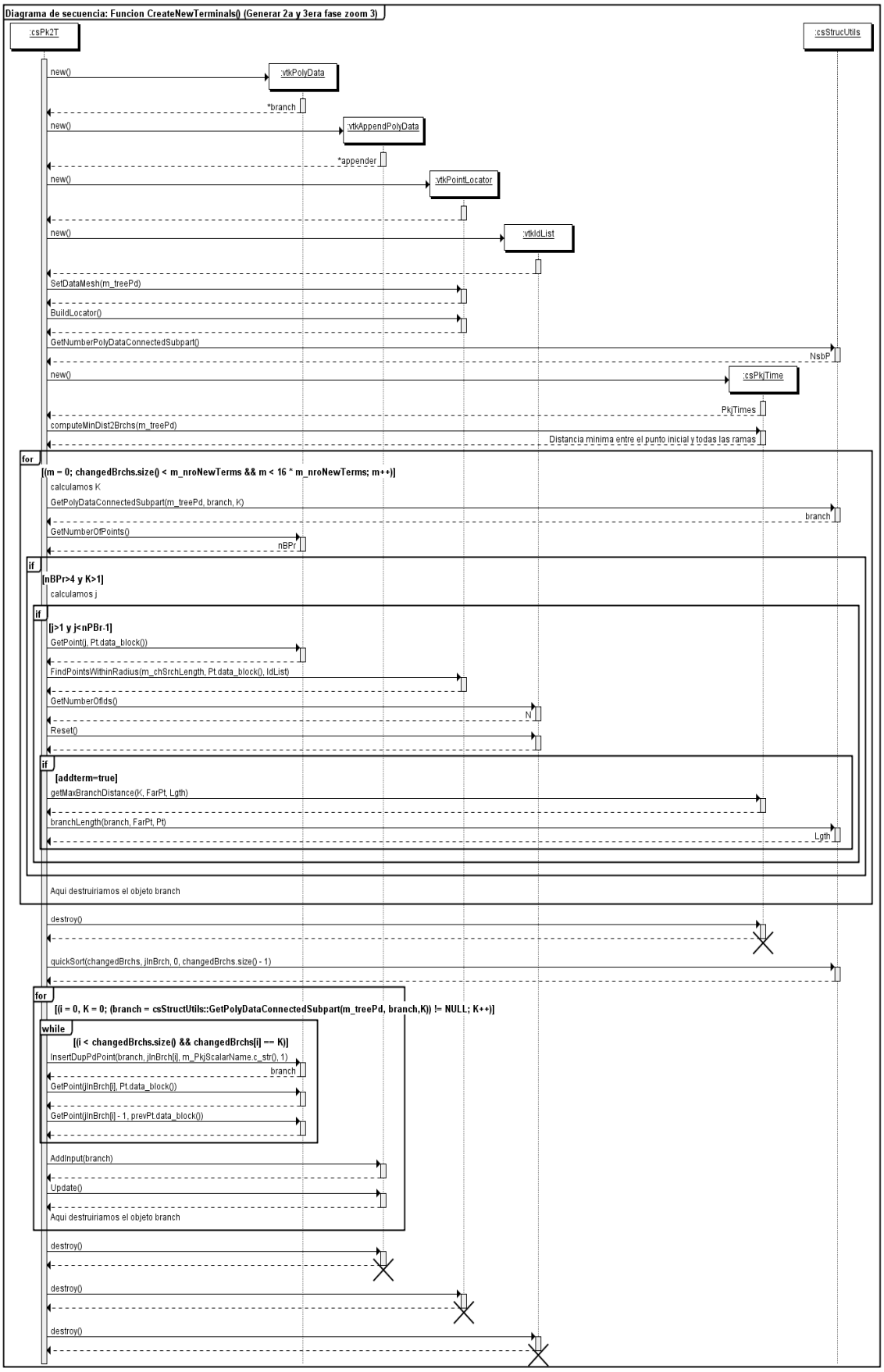

Figura 42. Diagrama de secuencia de createNewTerminals

Una vez creado los nuevos terminales en nuestra estructura, llamamos a "**Build1linePkjFlwAs2ndStep**". Construye la primera línea de ramas después de haber llamado a la clase csPkjT. Utiliza los terminales previamente definidos entre csPkjT y csPkj2T. Evita que las ramas se dirijan a la base del septum (fig. 43).

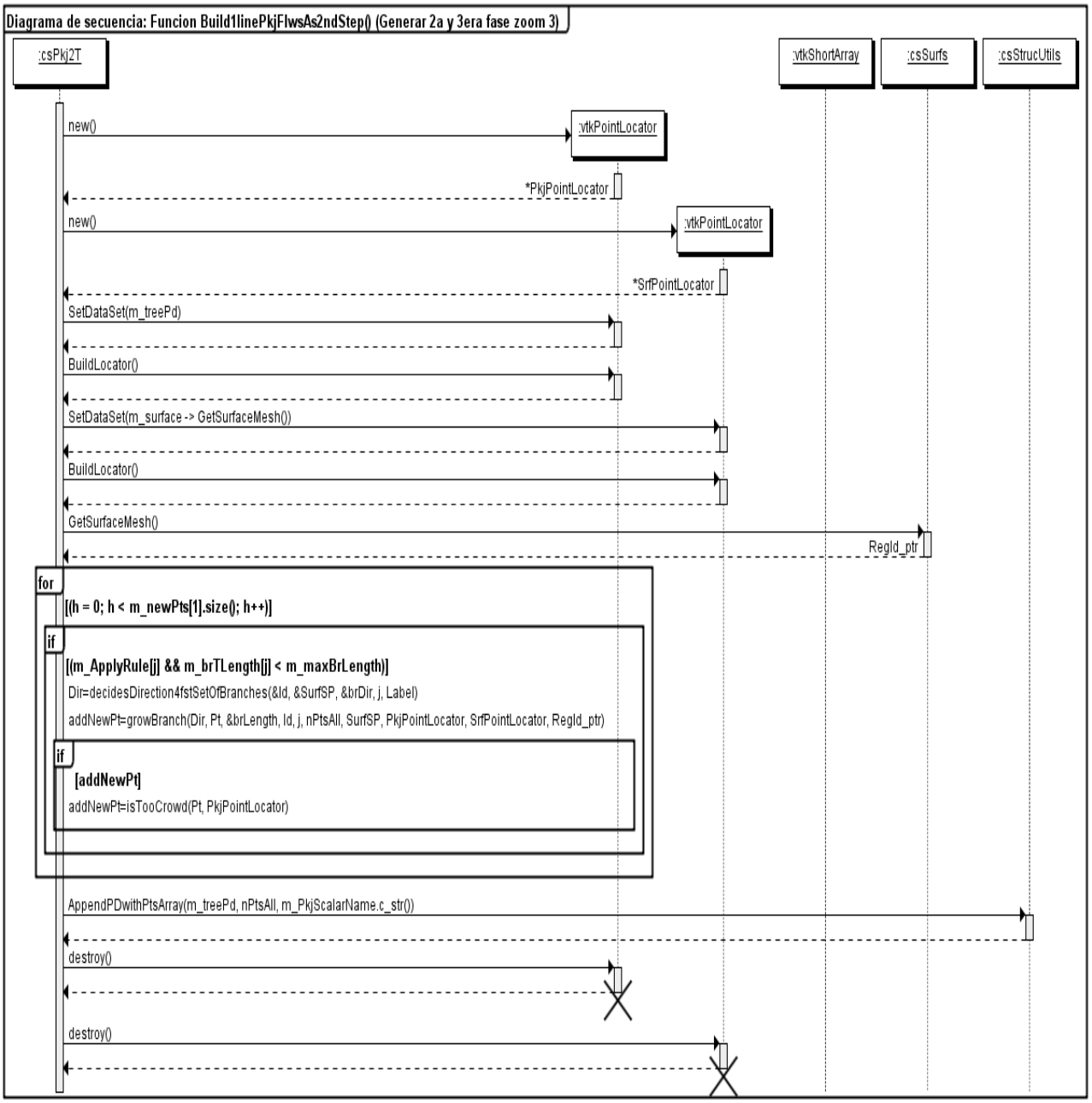

Figura 43. Diagrama de secuencia de Build1linePkjFlwAs2ndStep

Cuando ya hemos creado las primeras ramas de la segunda fase (o tercera fase), vamos a empezar a crear sub ramas a partir de estas. Para ello usamos "**newBranches**". En este método es donde vamos a explotar al máximo el *L-system*. Mediante una serie de reglas vamos a ir creando sub ramas a partir de las primeras ramas principales. Estas van creciendo (siempre respetando los parámetros de entrada) evitando los bucles pues hay un método específico para ello que veremos a continuación (fig. 44).

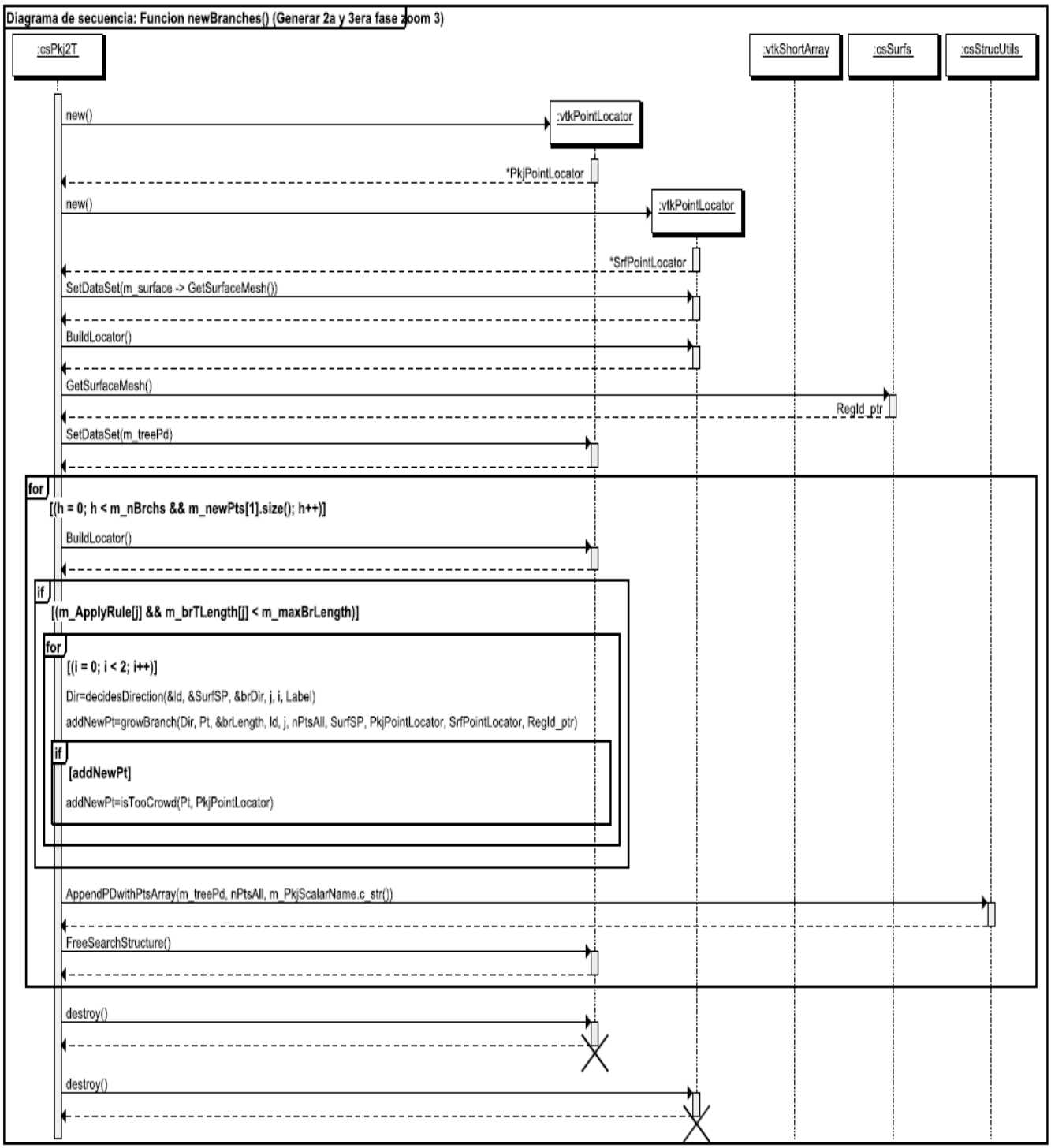

Figura 44. Diagrama de secuencia de newBranches

Para completar la estructura con todas sus ramas, hay que cerrar los posibles bucles que puedan existir. Hacemos esto porque, según los datos médicos que se han podido recopilar, se ha observado su existencia en la red de *Purkinje* real. Como nuestra idea es la fidelidad, hemos creado el método "**closeLoops**". Como es obvio por su nombre, está claro lo que hace. Crea puntos extra si son necesarios para unir las ramas en bucles.

Aparte de estos métodos, para generar la fase tres, hay que actualizar las ramas que hemos creado para indicar la primera y la última. Para ello tenemos la función "**updateFstLstBrchsNmbs**". Con esto lo que hacemos es preparar la estructura para iniciar la creación de la fase siguiente (fig. 45).

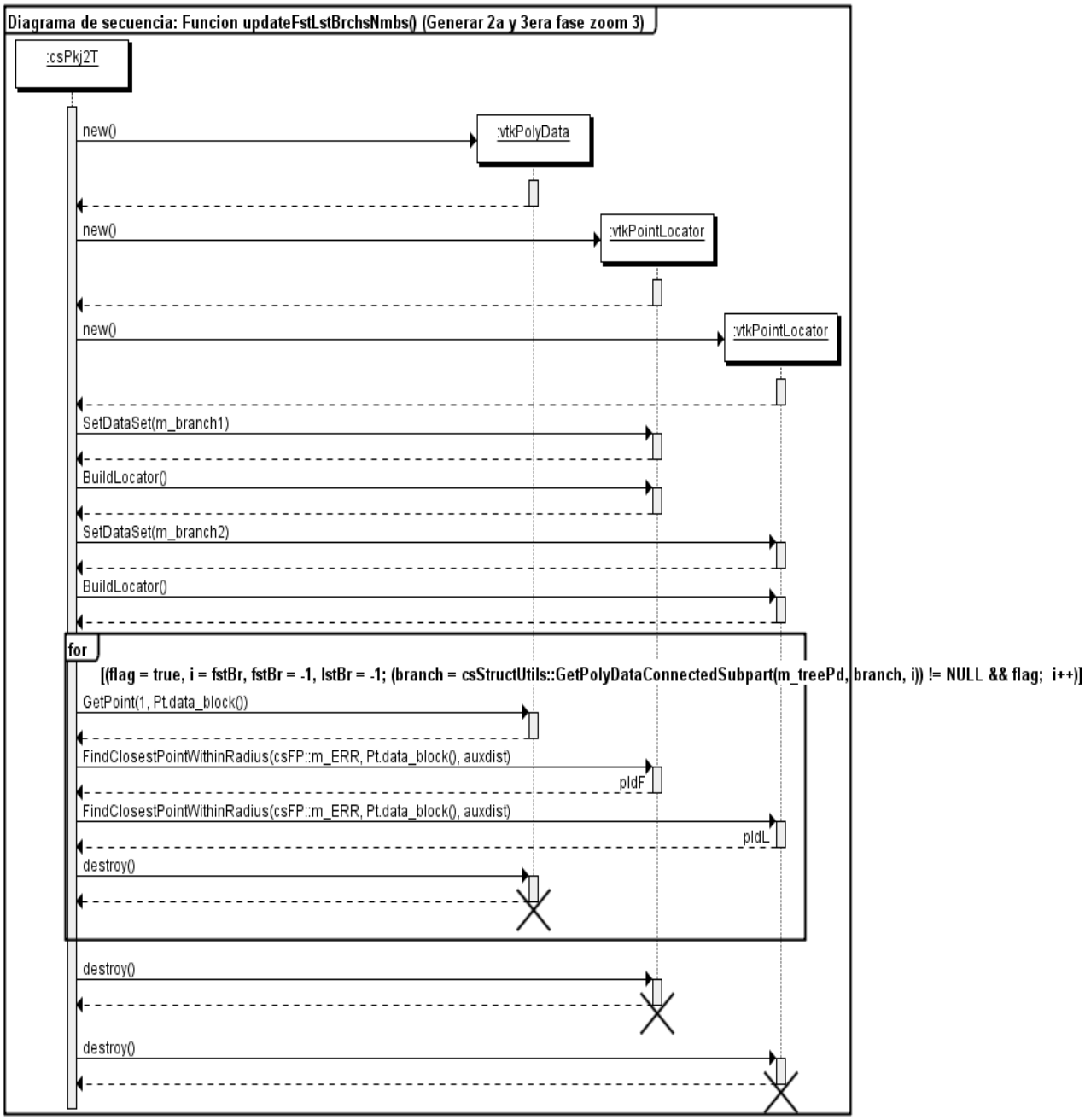

Figura 45. Diagrama de secuencia de updateFstLstBrchsNmbs

Junto a estos métodos tenemos otros específicos para su uso durante la generación de las fases dos y tres. El primero de dichos métodos es "**isTooCrowd**". Con ella comprobamos si el espacio alrededor de la nueva rama está muy poblado (fig. 46).

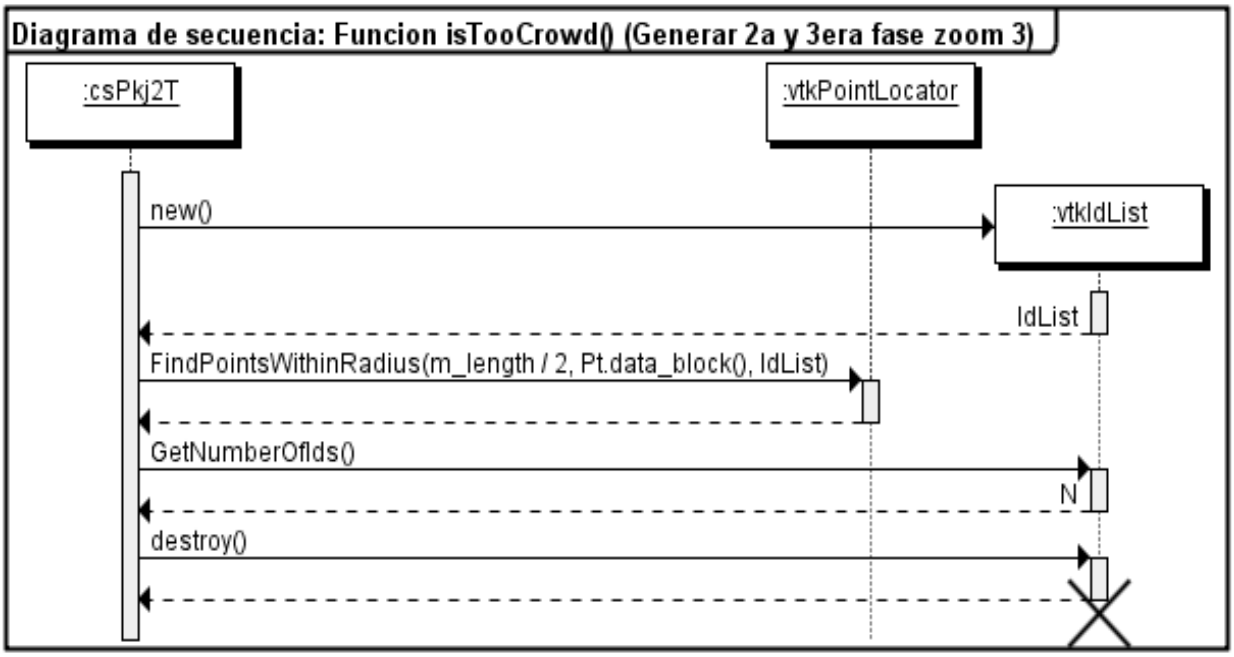

Figura 46. Diagrama de secuencia de isTooCrowd

Otra función nueva es "**checkNewTerminals**". Sirve para comprobar si el nuevo terminal que hemos creado está lo suficientemente apartado de los terminales que ya existen. Si no es así lo marca y no crece el árbol por ese punto. Si está en una situación propicia para una colisión con otra rama, se elimina.

"**deleteDupPts**" como su nombre indica nos permite eliminar puntos duplicados pero teniendo en cuenta que no crezca ninguna rama (fig. 47).

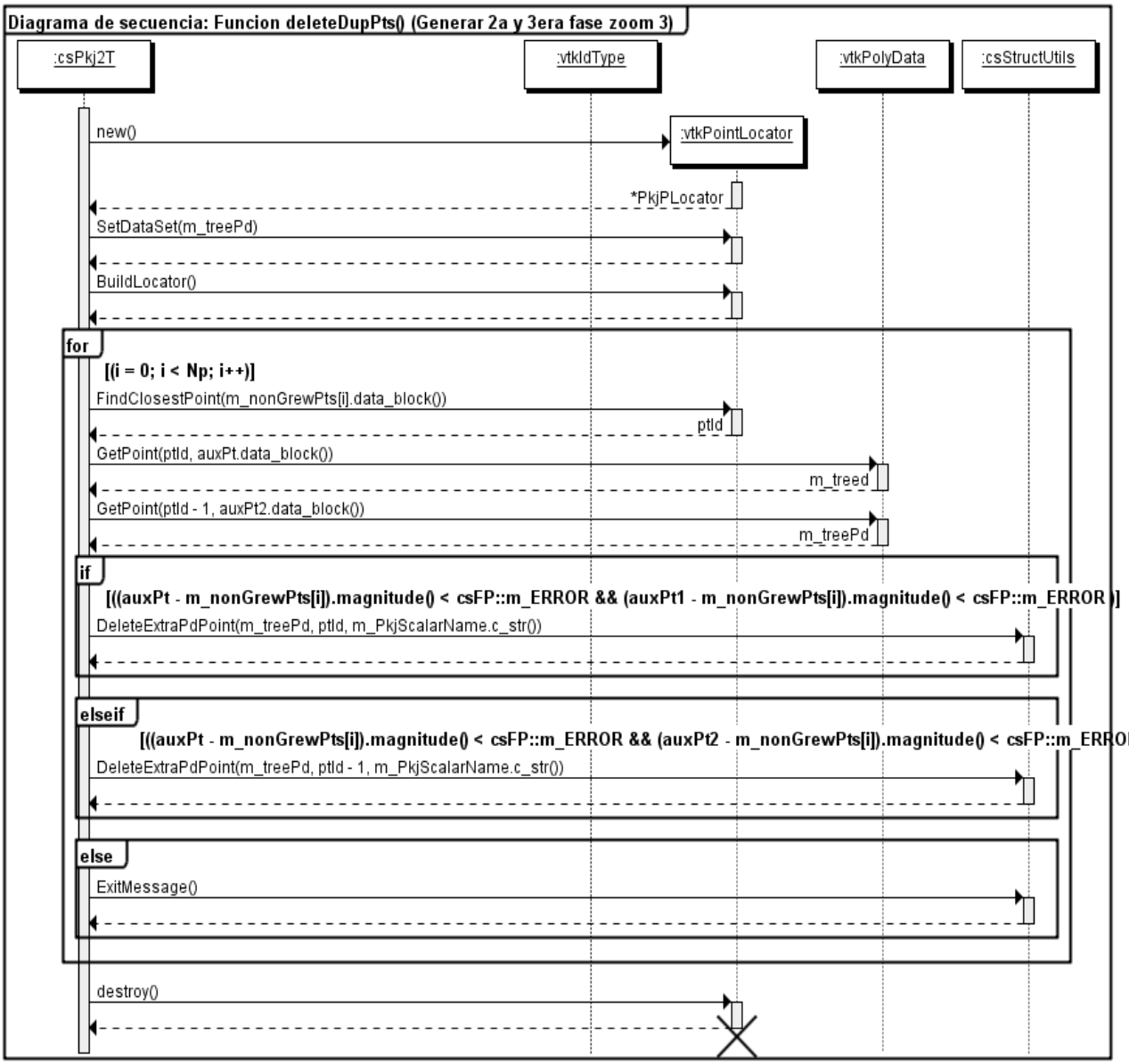

Figura 47. Diagrama de secuencia de deleteDupPts

El último método nuevo es "**decidesDirection4fstSetOfBranches**" y se encarga de calcular la dirección inicial de las nuevas ramas en la primera línea después de la fase uno (fig. 48).

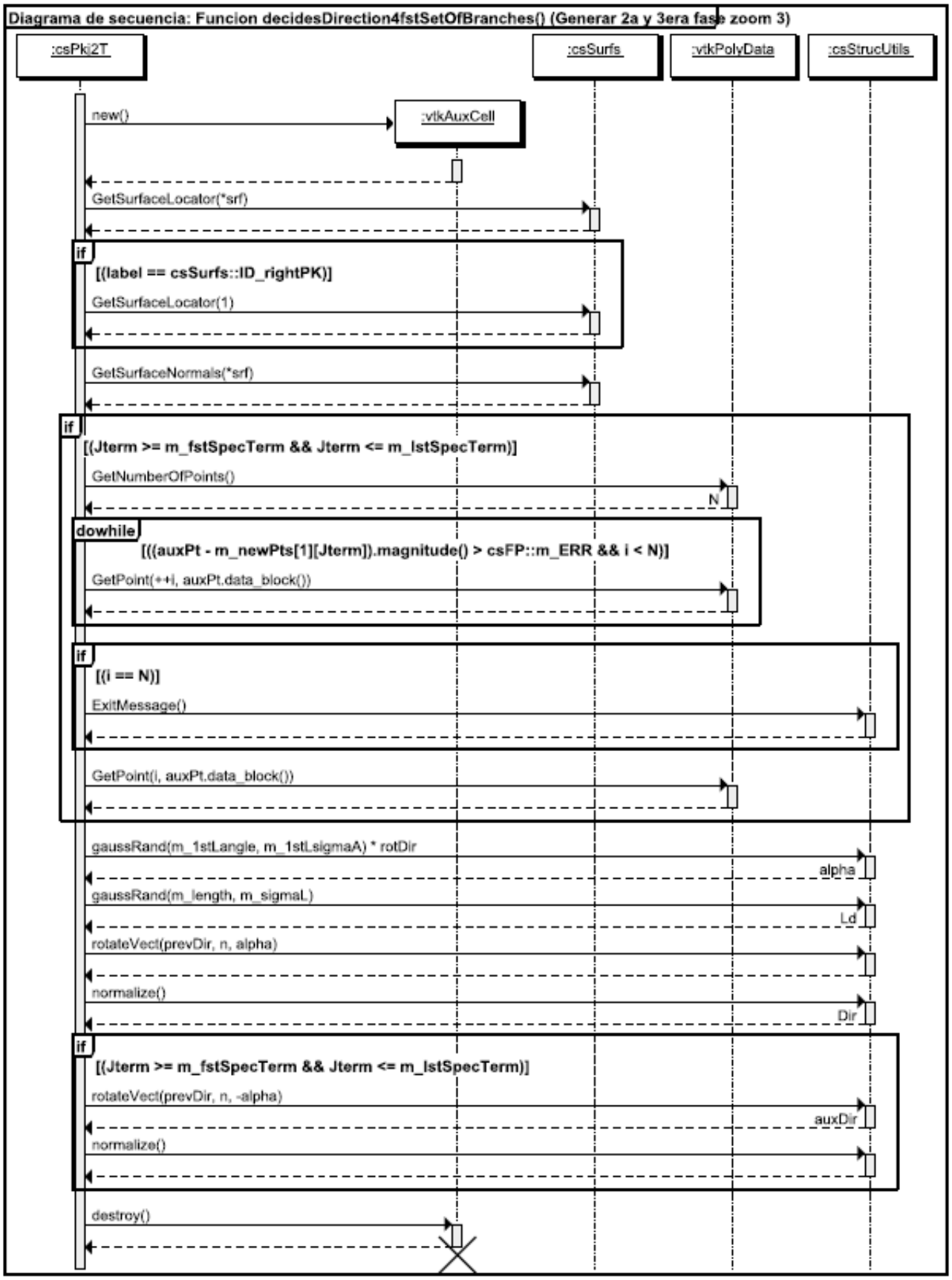

Figura 48. Diagrama de secuencia de decidesDirection4fstSetOfBranches

Junto con estos nuevos métodos, también tenemos otros que ya hemos usado durante la generación de la primera fase pero con ciertas modificaciones apropiadas a las nuevas características. No es necesario nombrarlas otra vez.

En los apéndices del presente documento (Apéndice.) expondremos las otras tres clases (csPkjTime, csSurfaces y csStructUtils) que hemos creado puesto que son clases de apoyo. Se deja al lector por si quiere satisfacer su curiosidad.

A continuación vamos a exponer como podemos usar las características de las clases para crear nuestros casos de uso en la interfaz gráfica.

5.3.2- Diseño de la interfaz gráfica

Una vez implementadas las diferentes clases que va a tener nuestra aplicación, hay que ver cómo vamos a enlazarlas para poder usarlas en la interfaz gráfica. Para ello primeramente crearemos el *plugin* general que será el encargado de controlar toda la aplicación. Después crearemos una serie de *widgets* que nos van a permitir ejecutar nuestros casos de uso.

Creación del plugin

GIMIAS se basa en la ejecución de una serie de *plugins* que se adaptan dentro de un entorno de trabajo genérico como ya vimos en el punto 5.2.2.

En nuestro caso vamos a generar uno desde cero al que se le puedan adaptar futuros *widgets*. Para ello vamos a utilizar una vez más la aplicación llamada “StartNewModule” (fig. 49).

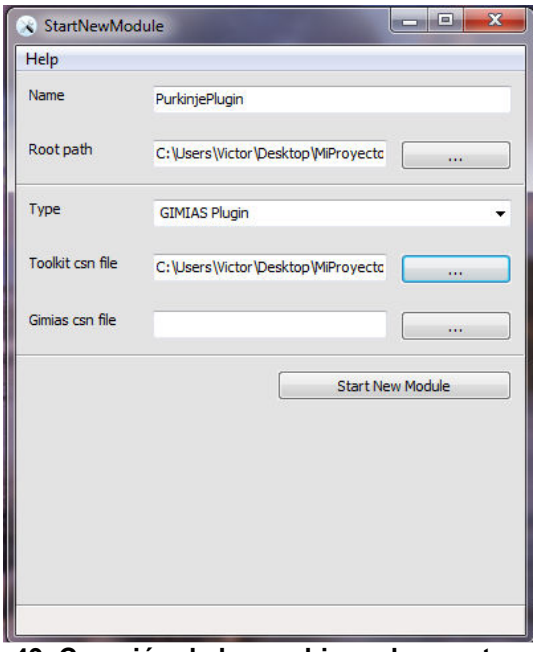

Figura 49. Creación de los archivos de nuestro *plugin*

Una vez ejecutado el programa, se nos van a crear las subcarpetas y los archivos (fig. 50) necesarios para empezar a crear nuestro *plugin* dentro del directorio *plugins* que hemos visto en la figura 31. Si ejecutásemos ahora nuestra aplicación, se nos generará sin problemas la solución del *plugin* pero al estar vacío no hará nada.

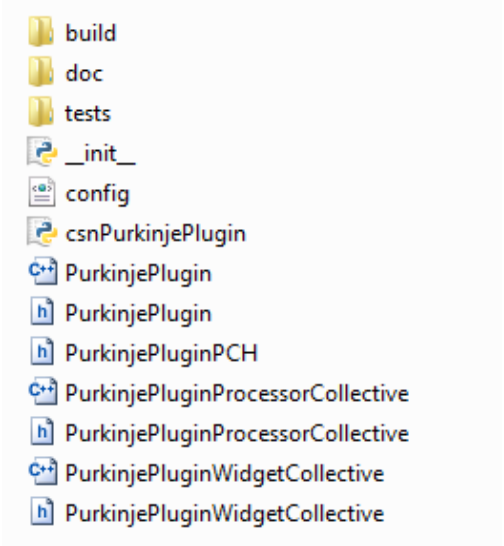

Figura 50. Subcarpetas y archivos de nuestro *plugin*

Una vez creadas las carpetas y los archivos generales del *plugin*, para añadirle funcionalidades vamos a crear diferentes *widgets* que nos permitirán ejecutar diferentes casos.

Creación de los widgets

Una vez más vamos a hacer uso de la aplicación "StarNewModule". Como podemos ver es muy sencilla y tiene un gran potencial que nos ahorra mucho trabajo. En este caso hay que marcar la opción *widget* (fig. 51).

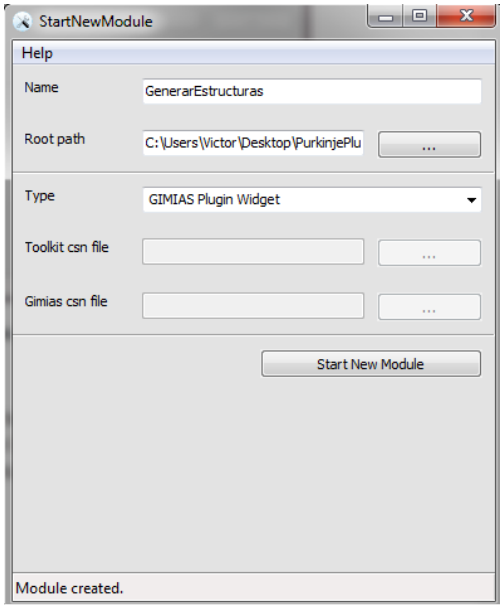

Figura 51. Creación de un *widget* para el *plugin*

Al ejecutar presionar sobre el botón *Start New Module* con las opciones configuradas como se ve en la figura, nos generará las subcarpetas y los ficheros para implementar nuestro nuevo *plugin* (fig. 52).

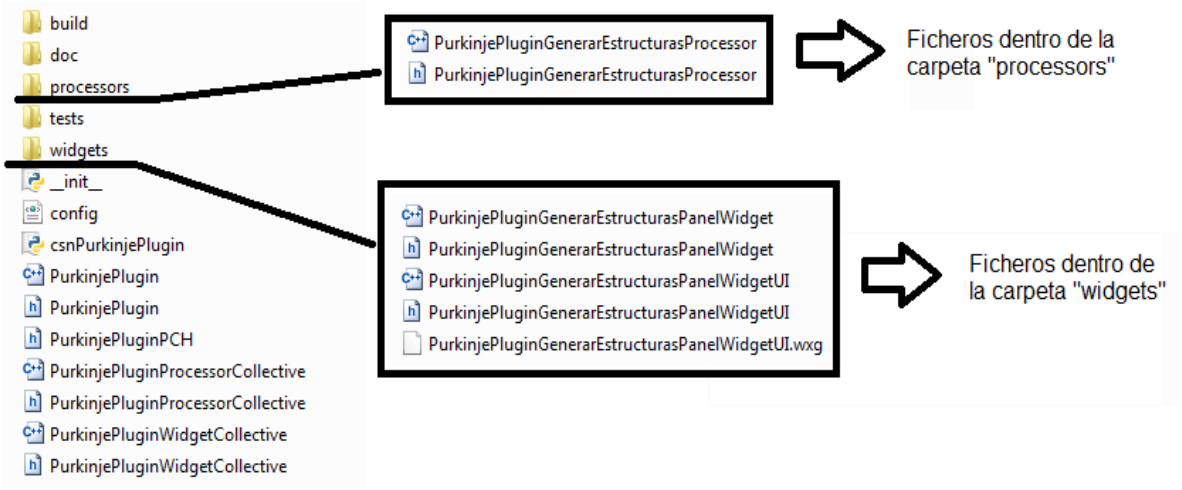

Figura 52. Estructura de las carpetas de los *widgets*

Como podemos ver, se generan una serie de ficheros divididos en dos carpetas nuevas: ***processors*** y ***widgets***. Si creásemos más *widgets*, se irían generando la misma cantidad de ficheros pero con nombres diferentes. Dentro de la carpeta ***processors*** tendremos el código del nuevo proceso del *widget* y dentro de la carpeta ***widgets*** está el código de panel del *widget* (la parte gráfica).

Una vez generados los ficheros anteriores, para diseñar como va a ser la parte gráfica del *widget* (el menú visual que vamos a anclar a nuestro *plugin*), vamos usaremos una aplicación implementada en python llamada wxGlade (referencia). Dicha aplicación nos va a permitir generar de forma sencilla parte del código de *plugin* pero en lo referente a los controles gráficos (botones, combox, textbox, etc…).

Al iniciar la aplicación se nos abre un menú como si fuese una aplicación de dibujo con su panel de opciones con múltiples botones. Abrimos el archivo terminado en wxg y se nos abre el panel gráfico del *widget* que posteriormente será el que acoplemos a nuestro *plugin*.

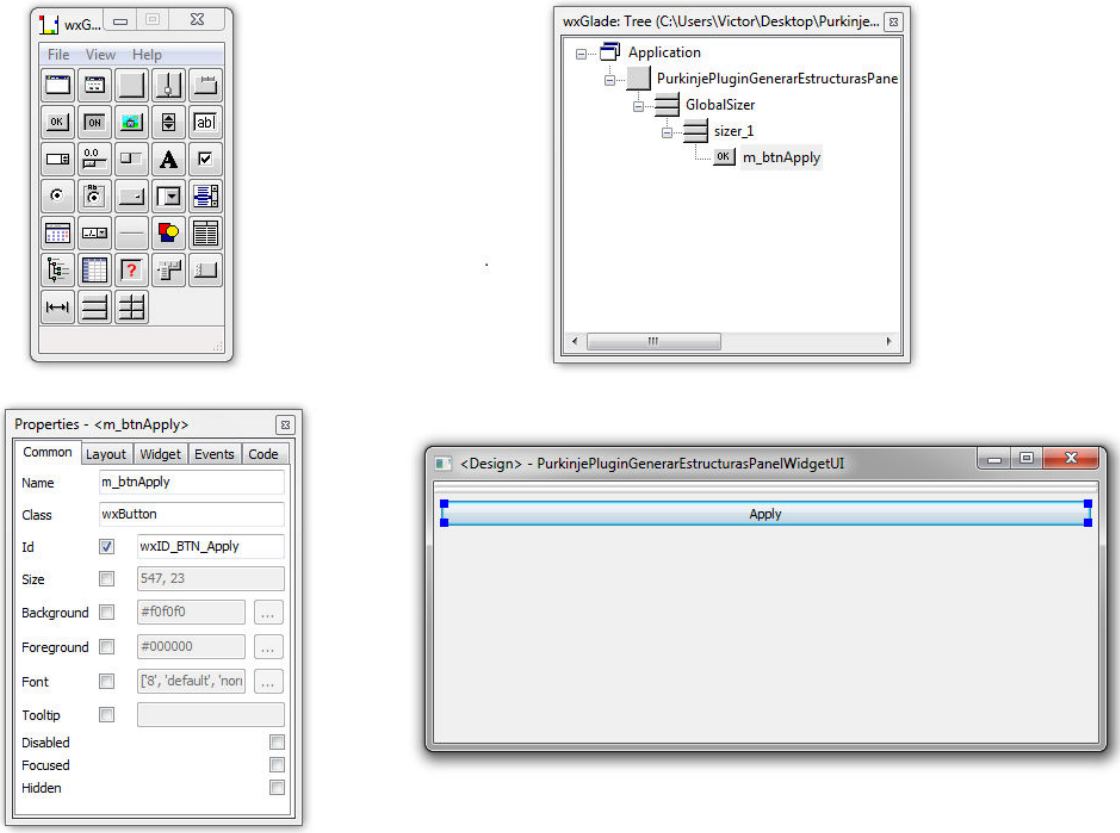

Figura 53. Modificación del panel del *widget* con wxGlade

Partiendo de aquí vamos a poder estructurar y modificar nuestro panel añadiendo o eliminando las opciones que queramos. Una vez finalizado el diseño, hacemos File->Generate Code (o Ctrl+G) y conseguiremos generar el código de la estructura gráfica con sus métodos. Posteriormente habrá que incluir el código necesario para que cada botón u opción añadida haga lo que queramos que haga.

Una vez descrito el proceso, vamos a ver como hemos generado nuestros *widget*s y cuál es su aspecto final.

Como hemos expuesto anteriormente, nuestra aplicación consta de dos casos de uso digamos más importantes. Por un lado tenemos la generación de la estructura original (es decir la generación normal de las fases), y por el otro una generación alternativa donde elegimos tres puntos para generar las tres primeras ramas.

Esto conlleva a estructurar nuestro *widget* en dos bloques que son:

*PurkinjePlugin*GenerateOriginalTreePanel*Widget* (fig.54)
*PurkinjePlugin*GenerateAlternativePanel*Widget* (fig. 55)

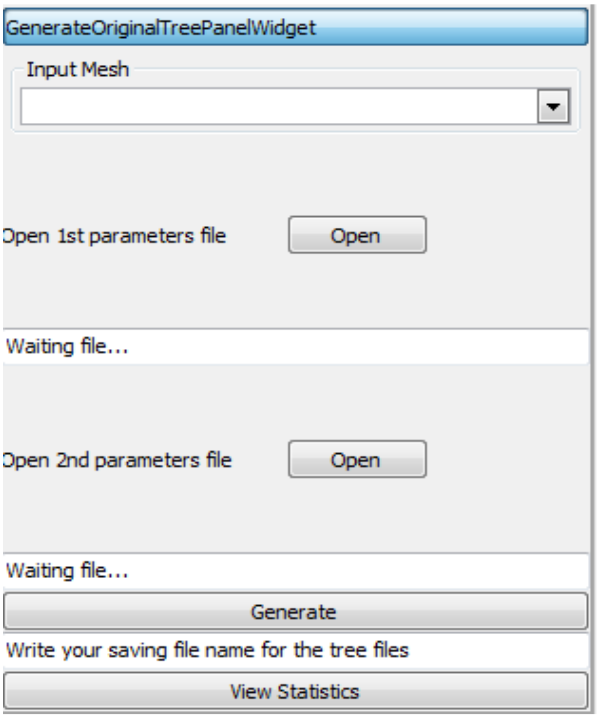

Figura 54. *PurkinjePlugin*GenerateOriginalTreePanel*Widget*

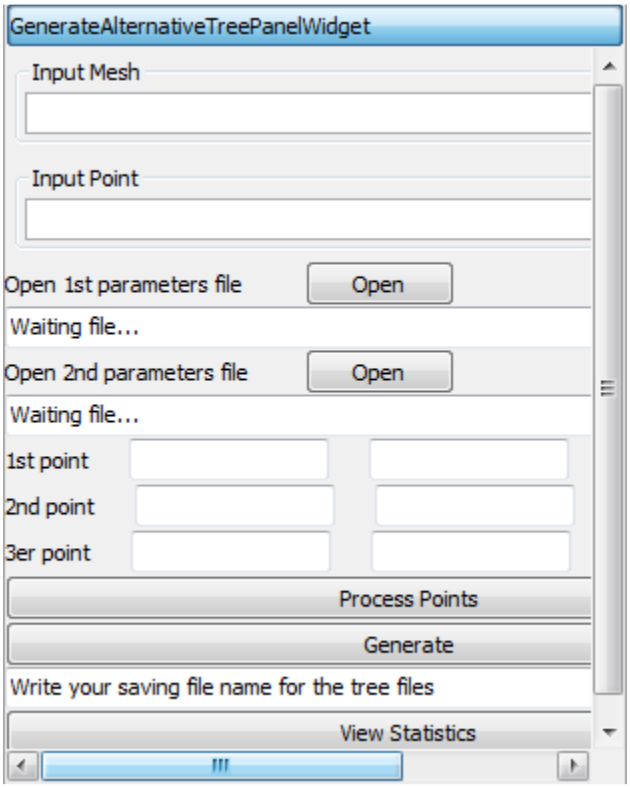

Figura 55. *PurkinjePlugin*GenerateAlternativeTreePanel*Widget*

Como podemos observar son bastante parecidos salvo el detalle de los recuadros de las coordenadas de los tres puntos seleccionados.

Vamos a detallar el funcionamiento de ambos paneles.

En el primero, al seleccionar la malla que vamos a tratar, nos saldrá su nombre en el recuadro “Input Mesh”, a continuación presionaríamos el primer botón de “Open” lo que provocaría que apareciese una ventana emergente con que nos permitirá navegar por los diferentes directorios para buscar nuestros ficheros de parámetros. La aplicación reconoce los ficheros txt como archivos de parámetros. Si por algún casual el usuario eligiese un fichero con otra extensión la aplicación lo detectaría y mostraría un mensaje de error como en la siguiente figura (fig. 56).

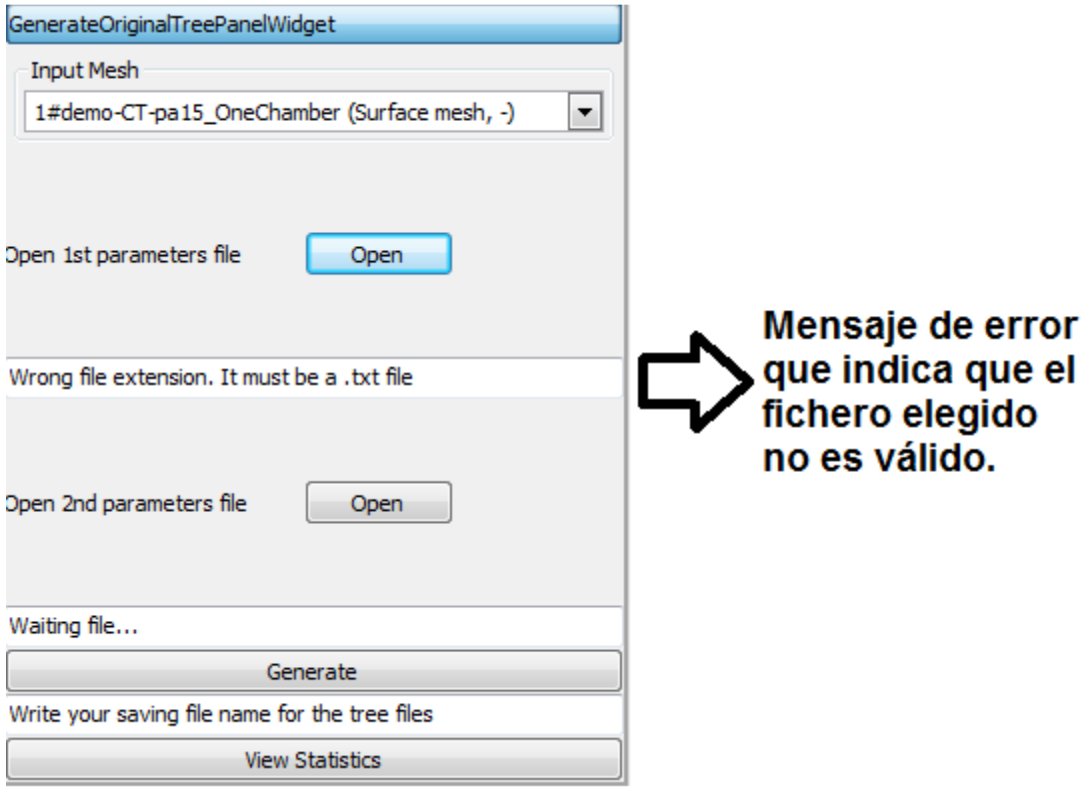

Figura 56. Error eligiendo el archivo con los parámetros

Una vez elegidos los dos ficheros con los parámetros necesarios para la generación de los árboles, escribimos un nombre característico para almacenar los ficheros que se van a generar. Esto se realiza en el recuadro que hay justo debajo del botón “Generate”. Cuando hemos realizado correctamente estos procesos, pulsamos sobre dicho botón el cual nos genera la estructura que podemos visualizar por pantalla (fig. 57). Hay que tener cuidado con el nombre que le ponemos para los ficheros que se van a generar puesto que se sobre escriben si ya están presentes en el disco duro.

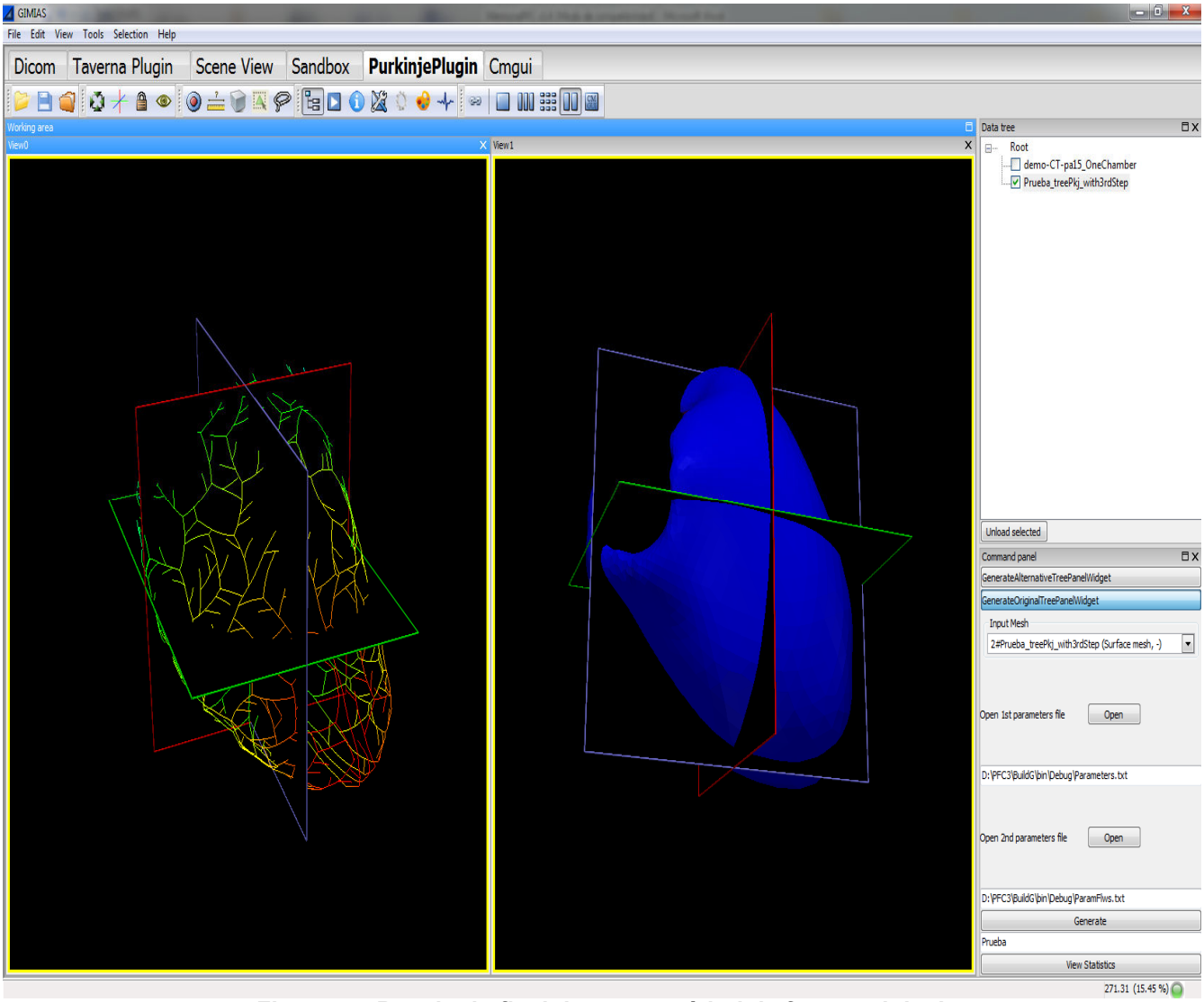

Figura 57. Resultado final de generar árbol de forma original

En esta imagen podemos ver el resultado final de generar el árbol de forma original. A la izquierda tenemos nuestro resultado. En el centro la malla sobre la que hemos calculado el árbol y a la derecha el panel de utilidades. En la parte superior un árbol con los objetos implicados y abajo nuestro panel.

El segundo panel funciona exactamente igual salvo por la introducción de los puntos sobre la malla. Para ello pulsamos sobre la herramienta de selección (fig. 58). Se nos abrirá un nuevo panel (fig. 59).

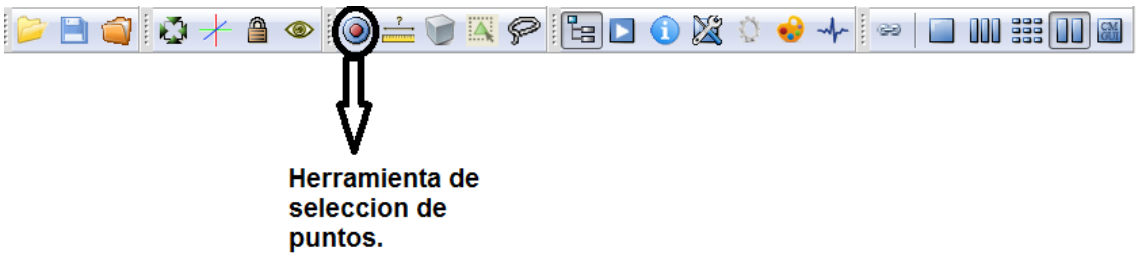

Figura 58. Acceso al panel de selección de marcas

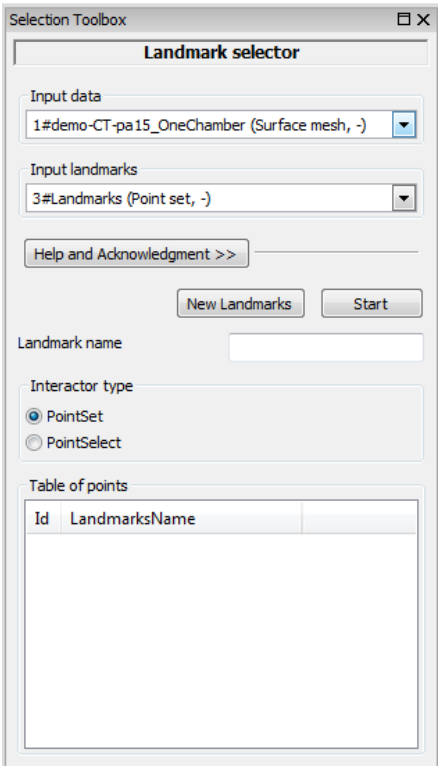

Figura 59. Panel de selección de marcas

A continuación podemos o bien pulsar sobre el botón “Start” o directamente hacer Shift+botón izquierdo del ratón donde queramos dejar una marca. Una vez finalizado (habremos puesto tres marcas), pulsamos sobre el botón “Stop” para terminar. Dichas marcas vienen con un nombre iniciar: “LandmarkX” donde “X” va incrementándose según el número que vayamos añadiendo (parte siempre de 0) (fig. 60).

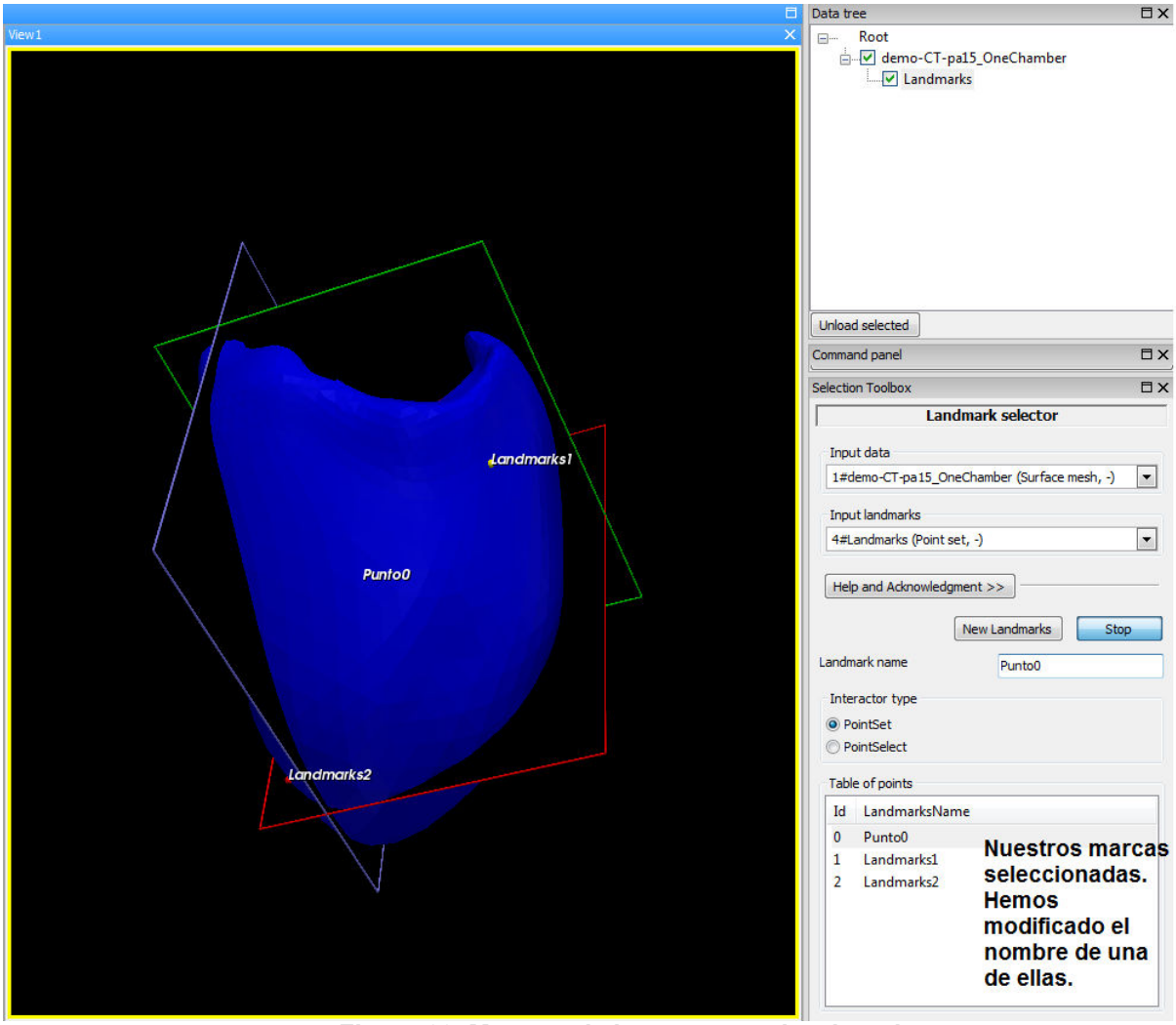

Figura 60. Muestra de los puntos seleccionados

Pero dicho nombre podemos variarlo según nuestro criterio. Nuestro siguiente paso pues, es el pulsar sobre el botón “Process Points” que nos procesara las tres marcas y nos devolverá sus coordenadas (fig. 61).

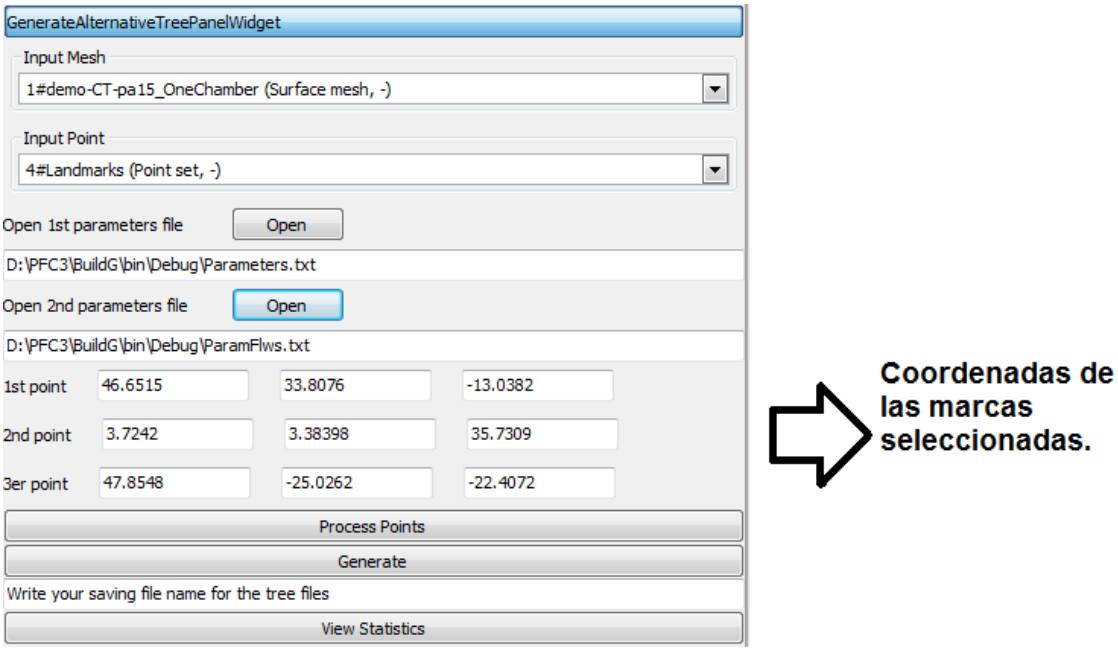

Figura 61. Coordenadas de los puntos seleccionados

El resto es igual, buscar y validar los ficheros de parámetros, escribir un nombre para los ficheros de salida y pulsar sobre el botón “Generate”. Una vez finalizado el proceso de generación, podremos ver el resultado por pantalla (fig. 62).

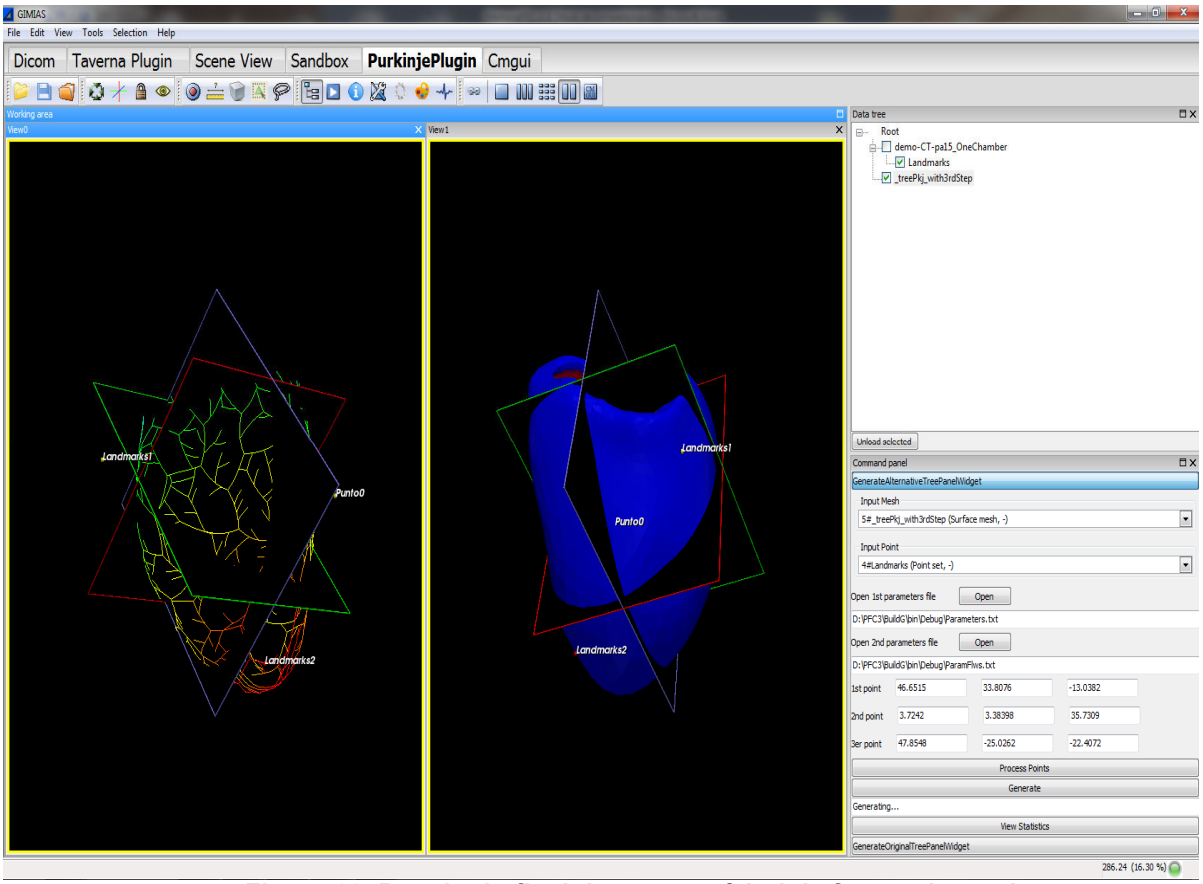

Figura 62. Resultado final de generar árbol de forma alternativa

A simple vista es posible que no se aprecie bien la diferencia entre el árbol generado originalmente y el árbol determinado por los tres puntos. Para ver más en detalle cómo afecta hemos preparado una serie de pruebas que expondremos en al punto 6 del presente documento.

5.3.3- Enlace y generación de la solución final

Una vez finalizado todo el proceso de implementación, hay que generar la solución para hacerla totalmente funcional. Para ello vamos a usar una herramienta llamada CSnake. El CSnake nos sirve para lanzar las aplicaciones necesarias para generar la solución de nuestro proyecto. Sirve de envoltura para otra aplicación denominada CMake. Ha sido creada por CISTIB usando el lenguaje python el cual provee un interfaz de usuario de alto nivel para los no usuarios de CMake, para poder configurar y generar soluciones como GIMIAS.

Esta aplicación es una plataforma-cruzada, *OpenSource* usada para controlar el proceso de compilación de software usando plataformas y archivos de configuración independientes del compilador.

Gracias a ella nos podemos olvidar de todos los archivos que habría que crear para enlazar todas las clases entre ellas.

Aparte de estos dos programas también vamos a necesitar el Microsoft Visual Studio 2008 y un intérprete para python.

Una vez instalado todo, abrimos la aplicación CSnake para empezar a configurar las opciones de compilación (fig. 63).

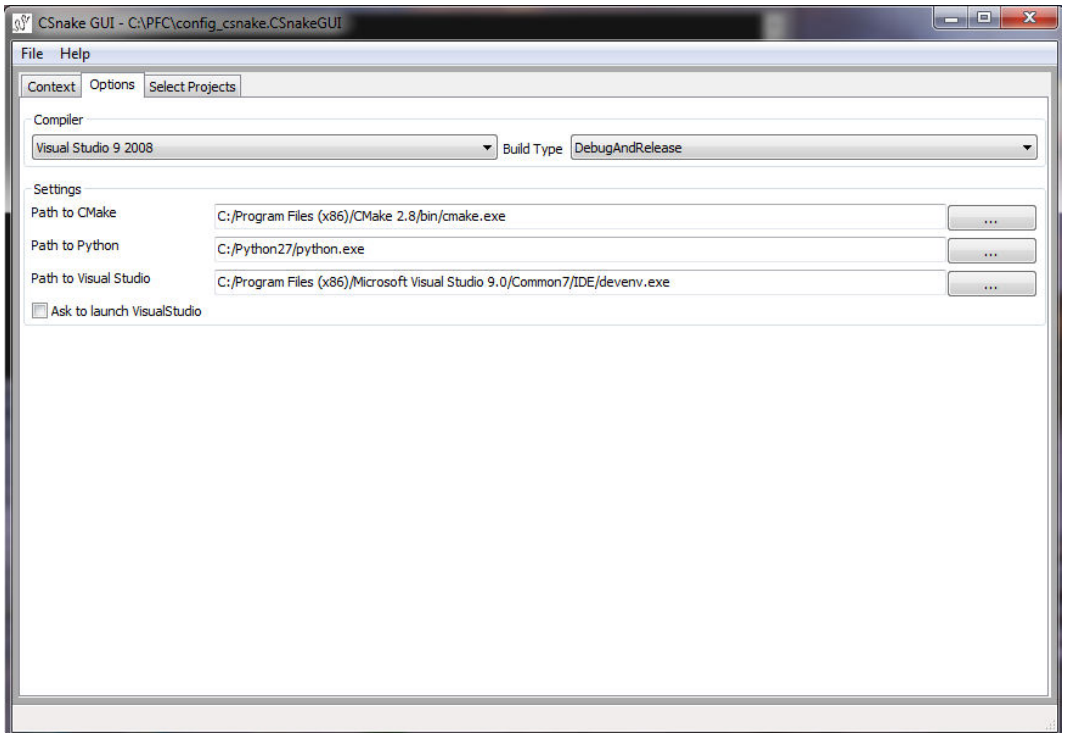

Figura 63. Configuración de las opciones de compilación en CSnake

Una vez seleccionadas las opciones generales, hay que generar GIMIAS desde el código fuente. Esto puede llevar un poco de tiempo dependiendo del PC donde se realice.

Cuando lo hayamos generado (o si ya lo estuviese) pasamos a compilar nuestras librerías. Cuando hemos generado los archivos con la aplicación "StartNewModul", uno de los ficheros que se generan es un python (csnMyProject*Toolkit*.py) que nos indica cual es el contenido de nuestro proyecto en cuanto a librerías y/o *plugin*s (fig. 64)

```
import csnBuild
import csnCilab

def purkinjeLib():
    import modules.PurkinjeLib.csnPurkinjeLib
    return modules.PurkinjeLib.csnPurkinjeLib.purkinjeLib
```

Figura 64. Archivo csnProject*Toolkit*.py

Este fichero es el que nos indica las librerías que constituye nuestro proyecto. Gracias a él podemos, mediante el CSnake enlazar todos los archivos para poder generar la solución completa.

Posteriormente, hay que editar el archivo python de nuestra librería (csn*Purkinje*Lib.py) que sirve para enlazar nuestra librería con la aplicación generar así como añadir las librerías *ThirdParty* necesarias (fig. 65).

```
# Used to configure PurkinjeLib
import csnCilab
from csnToolkitOpen import *

purkinjeLib = csnCilab.CilabModuleProject("PurkinjeLib", "library")
purkinjeLib.AddLibraryModules(["uStruct"])
purkinjeLib.AddProjects([baseLib, vtk, boost, zlib, baseLibVTK, toolkit])
# purkinjeLib.AddTests(["tests/tlFirstTest/*.*"], cxxTest)
purkinjeLib.SetPrecompiledHeader("PurkinjeLibPCH.h")
```

Figura 65. Archivo csn*Purkinje*Lib.py

Hay que tener en cuenta que nuestro proyecto es externo a GIMIAS por lo que cuando vayamos a compilarlo, hay que enlazarlo con el *framework* para poder usarlo posteriormente (fig. 66).

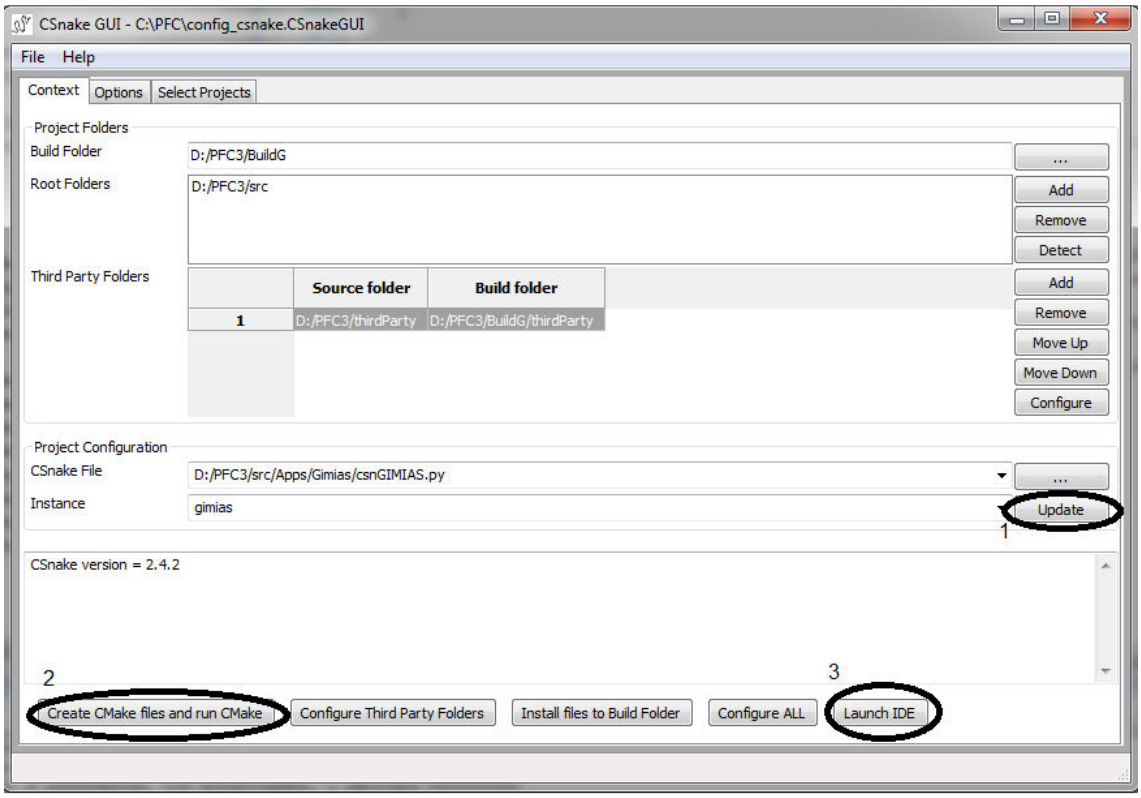

Figura 66. Compilación de nuestras librerías.

Cuando pulsemos sobre *Launch IDE*, se nos abrirá el Visual Studio. Lo que haremos a continuación es generar la solución de nuestra librería para comprobar que no haya errores en el código.

Finalizado este primer proceso, a continuación vamos a compilar nuestro *plugin* y añadirlo al módulo general. Para ello al crear nuestros ficheros de *plugin*, se nos generará uno, que en nuestro caso se llama *csnPurkinjePlugin* (fig. 67).

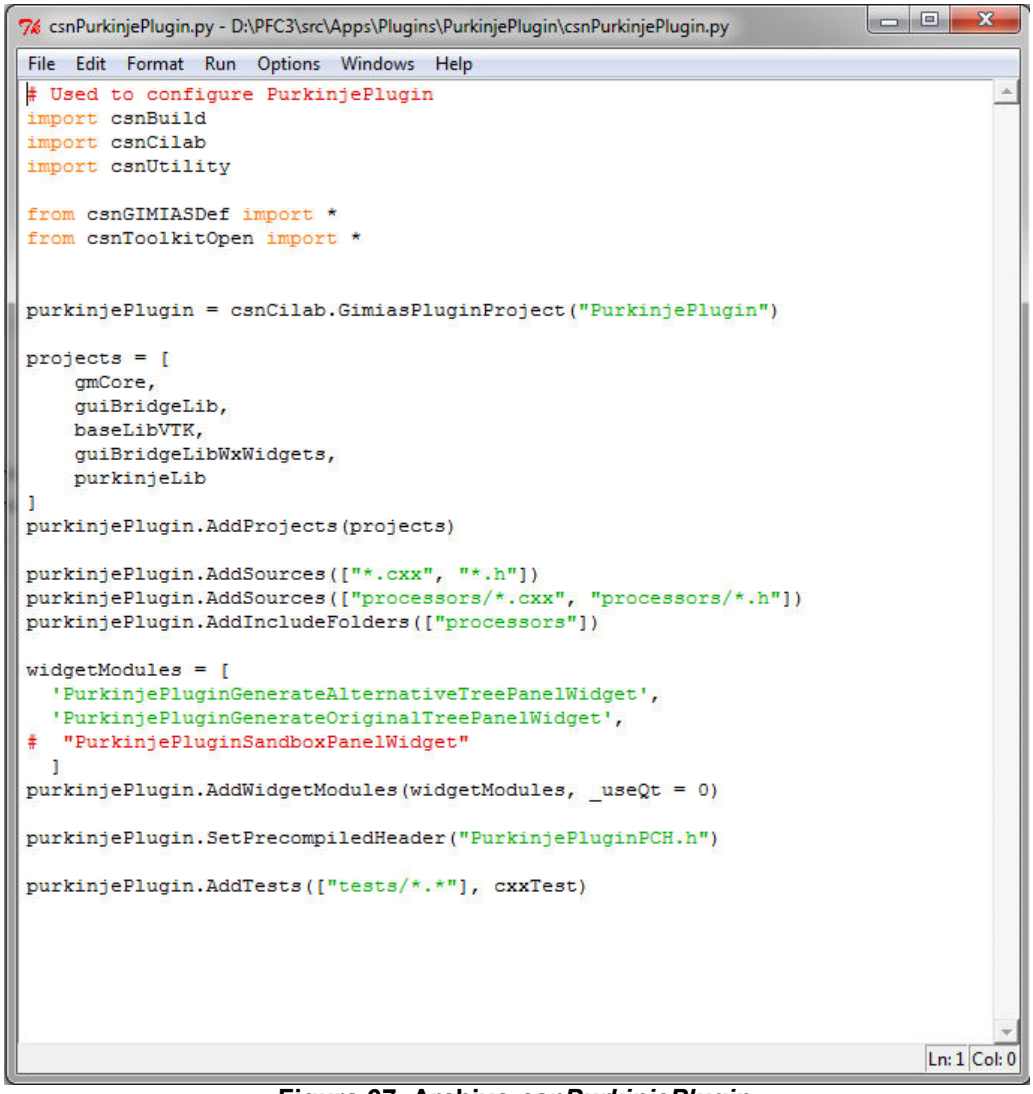

Figura 67. Archivo *csnPurkinjePlugin*

En este fichero, es donde se muestra el *plugin* de nuestro proyecto así como los *widgets* que tiene. Una vez comprobado que todo es correcto, abrimos el CSnake y lo configuramos para poder compilar el *plugin* dentro de GIMIAS (fig. 68).

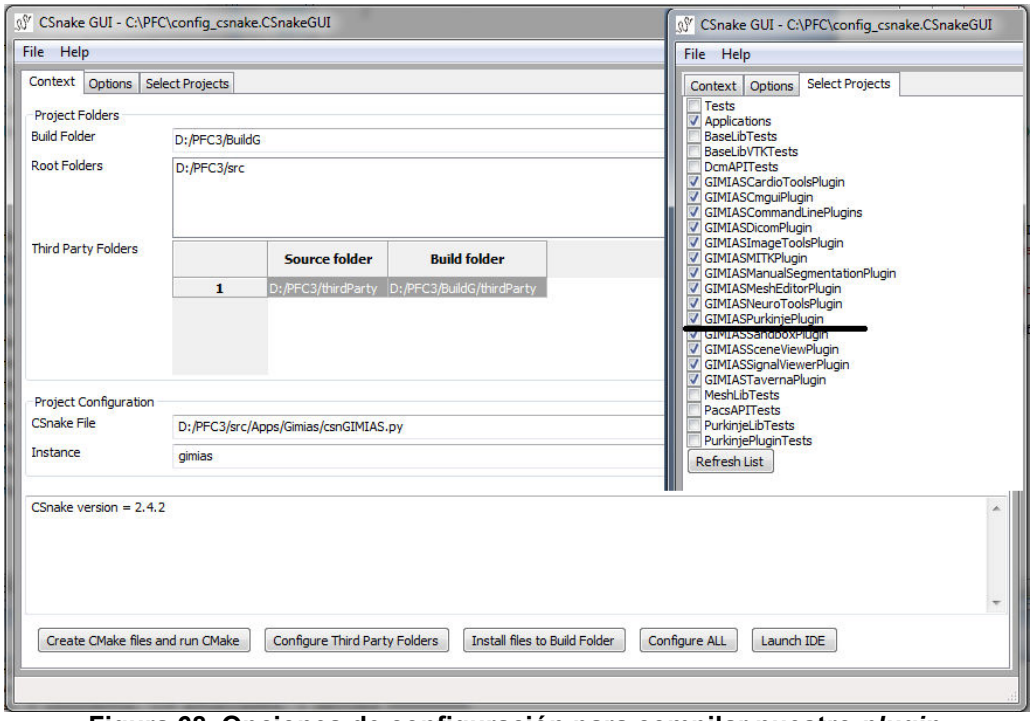

Figura 68. Opciones de configuración para compilar nuestro *plugin*

Al pulsar en *Launch IDE*, se nos abrirá el Visual Studio con todos los archivos de la aplicación final lista para generarla (fig. 69).

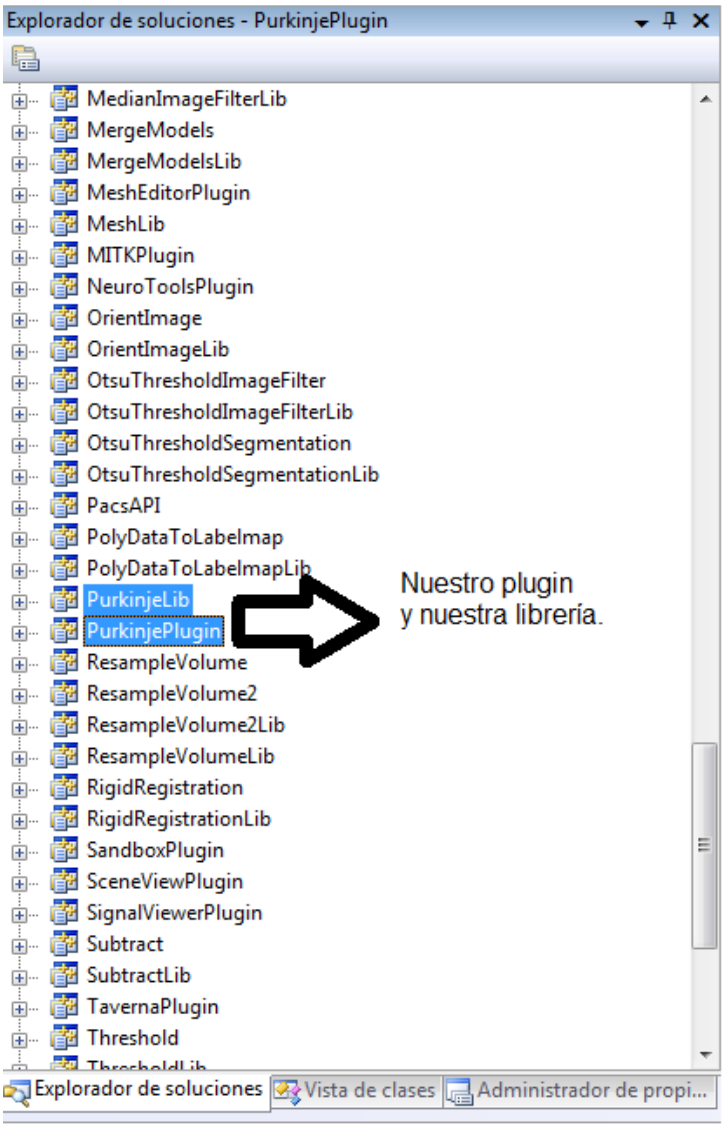

Figura 69. Archivos de la solución final

Como podemos ver, nuestra librería y nuestro *plugin* están totalmente integraos en la solución final de GIMIAS. Ahora es cuando podemos empezar a implementar nuestro código y hacer uso, si fuese necesario de las herramientas que se nos ofrece en la aplicación base.

En la siguiente figura (fig. 70) vamos a ver de forma general el número de ficheros que disponemos para codificar nuestro *plugin*.

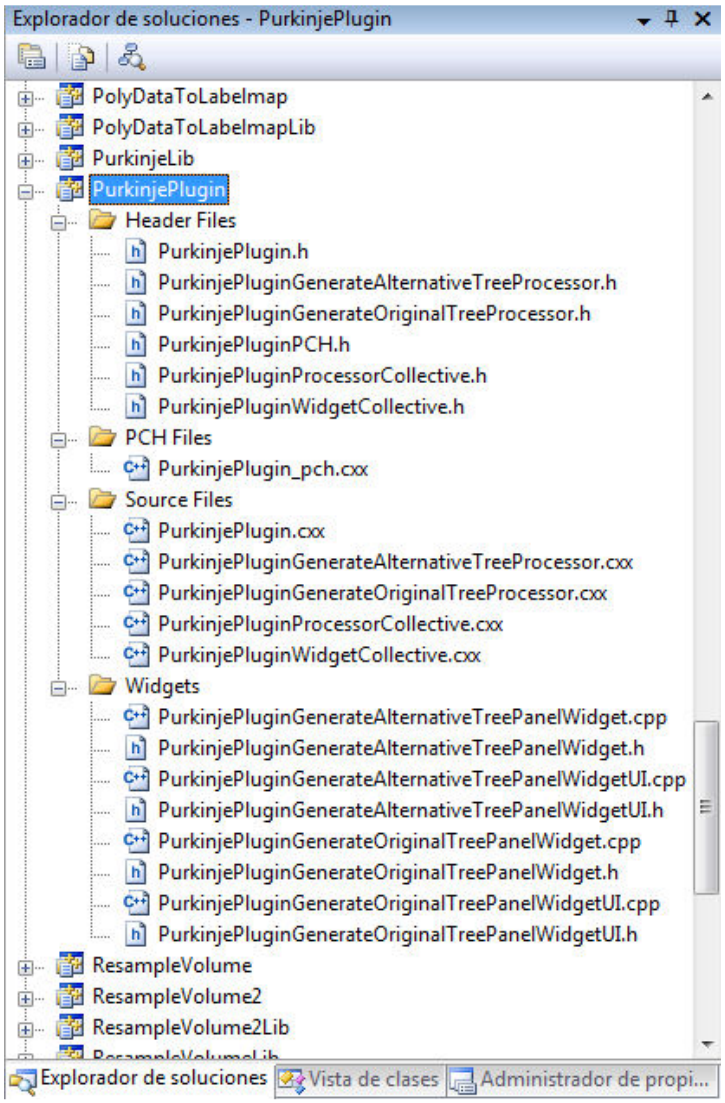

Figura 70. Los ficheros de nuestro *plugin*

Una vez integrado en GIMIAS, para poder hacer uso de los métodos de nuestras clases, hay que llamar mediante la opción "#include" una serie de librerías dentro del código de nuestro *plugin*. Gracias a esto, enlazaremos nuestra librería con el *widget* y este con GIMIAS. Generamos la solución entera y obtendremos el ejecutable de GIMIAS, el cual llevará integrado nuestro *plugin* para usarlo siempre que queramos.

5.3.4- Codificación

En este punto vamos a explicar, intentando no entrar en mucho detalle, como hemos implementado, a nivel de código, los *widget*s que hemos creado.

En la figura anterior, podemos ver los diferentes archivos que hemos creado. Vamos a explicar los **Widget*UI.h/cxx, **Widget*.h/cxx y los *Processor.h/cxx.

******Widget***UI.h/cxx:**

En estos ficheros va a estar toda la parte de código referente a la interfaz gráfica del *widget* (de ahí su terminación en "UI"). Con esto queremos decir que tendremos el código para dibujar las cajas de texto, los títulos, los botones así como sus posiciones y/o distribuciones. Estos ficheros se generan con la aplicación citada anteriormente ("StartNewModule") pero se rellenan con el programa wxGlade (que ya hemos explicado anteriormente). Una vez rellenados con la información de que herramientas constituyen el *widget* y su posición ya no lo tocaremos más pues no será necesario.

******Widget***.h/cxx:**

Aquí vamos a implementar los métodos necesarios para poder tratar la información que nos entra directamente cuando interactuamos con nuestra interfaz gráfica. Es decir que hay que mostrar en las cajas de texto o que hay que hacer cuando se pulsa un determinado botón de la interfaz. En nuestro caso al tener diferentes botones y cada uno hace una acción determinada, vamos a tener un método para cada botón. Su función principal es la de controlar y mostrar al usuario que sucede cuando interactuamos con un determinado elemento de la interfaz gráfica. Los métodos aquí presentes llamarían a otros métodos del siguiente nivel (los ficheros *Processor.h/cxx) para realizar los algoritmos pertinentes. Esto se consigue con la declaración de un puntero al fichero *Processor.cxx. Con este puntero tendremos acceso a los métodos superiores.

***Processor.h/cxx:**

Para finalizar tenemos el grupo de ficheros *Processor.h/cxx. Estos ficheros son los más importantes puesto que van a ser los que procesen la información introducida por el usuario mediante la interfaz gráfica y la gestiona según nuestro deseo. Por ejemplo cuando pulsamos sobre el botón "Open" para seleccionar el fichero de parámetros, el fichero **Widget*.cxx tiene el método OnOpenFstParamFile() que llamará a su vez al método Update(int opción) que está presente en el fichero *Processor.cxx.

En estos ficheros será donde llamemos a los métodos de nuestra librería para poder generar los diferentes árboles.

Con nuestra solución ya creada y funcional al 100%, vamos a realizar una serie de experimentos para comprobar si todo funciona como lo esperamos y si los resultados obtenidos nos van a servir para un posterior estudio.

6- Pruebas y resultados

A continuación vamos a realizar una serie de experimentos para comprobar las diferencias en los resultados que nos aportan los dos árboles diferentes.

6.1- Descripción de experimentos

Hemos realizado un banco de pruebas modificando ciertos detalles dentro del fichero de los parámetros de la segunda fase de la generación del árbol de *Purkinje*. Hemos ido modificando el valor de la longitud de las ramas (0.5, 1, 2, 3, 4 mm), el número de ramas que se crean después de la primera fase (64/128) y el número de ramas que se generan en la tercera fase (128/256). Además para la generación de árbol alternativo hemos tomado tres puntos cuyas coordenadas son:

Punto1: x -> 52.4066, y -> 35.1956, z -> 2.61383
Punto2: x -> 16.0918, y -> 45.1776, z -> 35.2354
Punto3: x -> 33.0906, y -> 48.29, z -> -10.993

Hemos tomado siempre los mismos tres puntos puesto que como la selección la hacemos nosotros y puede no ser del todo precisa, no deseamos que haya mucha variabilidad entre los resultados.

Todo esto nos ha dado lugar a 20 árboles diferentes (10 para la generación original y 10 para la alternativa).

Lo que nos interesa estudiar a continuación es ver cómo pueden llegar a influir estos valores en el número de ramas que se crean, de forma aleatoria, en cada una de las 17 regiones en las que hemos dividido el corazón. Para ello vamos a observar los valores mediante una serie de gráficos.

6.2- Resultado y discusión

Una vez realizado los bancos de pruebas, podemos acertar si decimos que nuestro software cumple con los requisitos propuestos por el cliente. Es capaz de generar una red de *Purkinje* totalmente funcional sin restricciones. La generación de las mallas es, en nuestros casos de pruebas, relativamente rápida tardando como mucho 10min. Pero esto dependerá en gran medida de los parámetros que le introduzcamos por los ficheros de texto. A mayor número de ramas a generar más tardará puesto que se tendrían que realizar muchos más cálculos para comprobar todas las peculiaridades que hay que tener en cuenta. A continuación vamos a poner una serie de imágenes más detallas de nuestra red de *Purkinje* para comprobar si los resultados finales son parecidos a los reales.

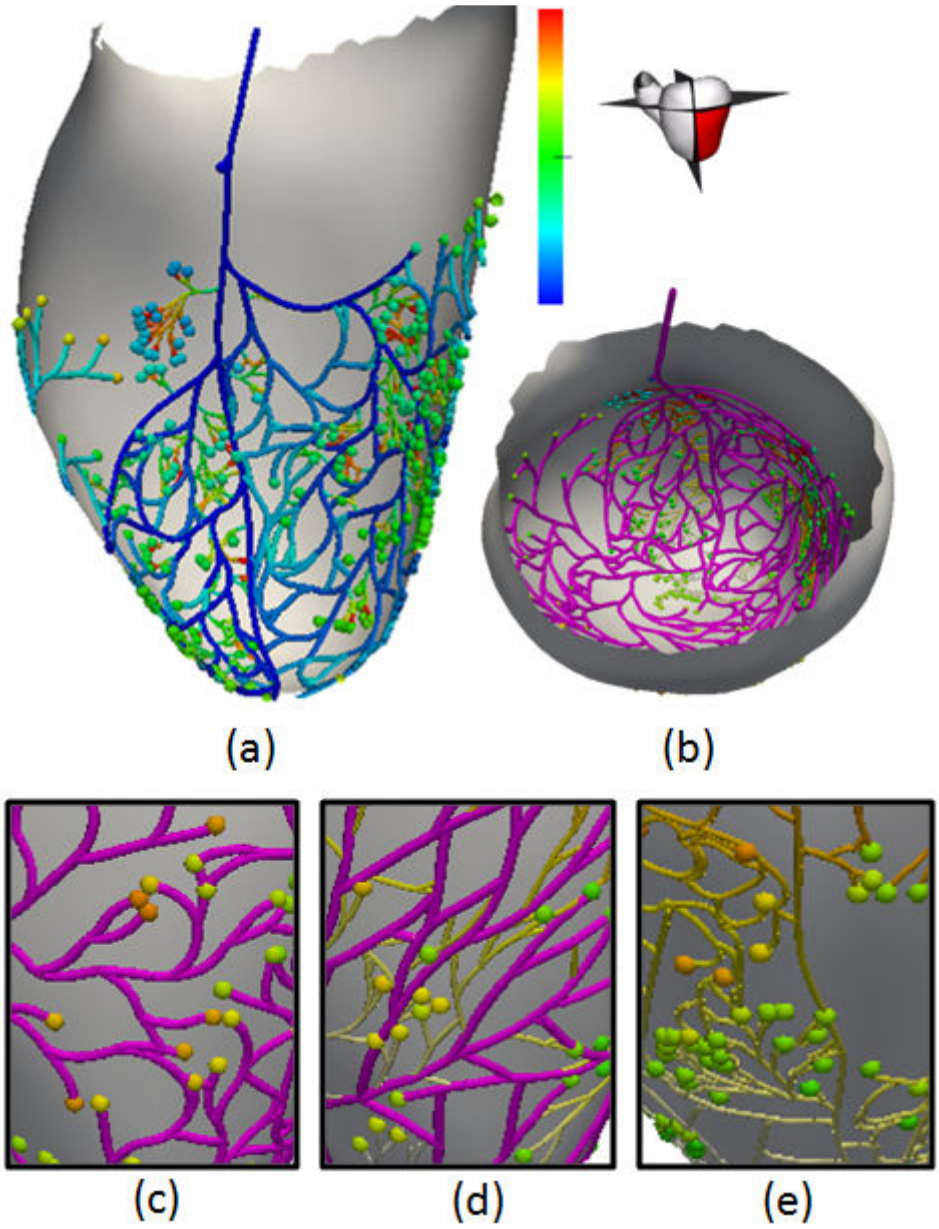

Figura 71. Red de *Purkinje* 3D del ventriculo izquierdo. a) Vista transversal. b) Vista superior. c), d) y e) Detalles más localizados

En esta imagen (fig. 71) podemos ver el ventrículo izquierdo desde diferentes ángulos y posiciones. Vemos de forma general como queda la estructura del árbol y como se expande hasta alcanzar las paredes del corazón. Las esferas pequeñas representas los puntos donde las ramas “tocan” la estructura.

A continuación vamos a ver una serie de imágenes más detalladas e individuales representando varias fases del proceso de generación.

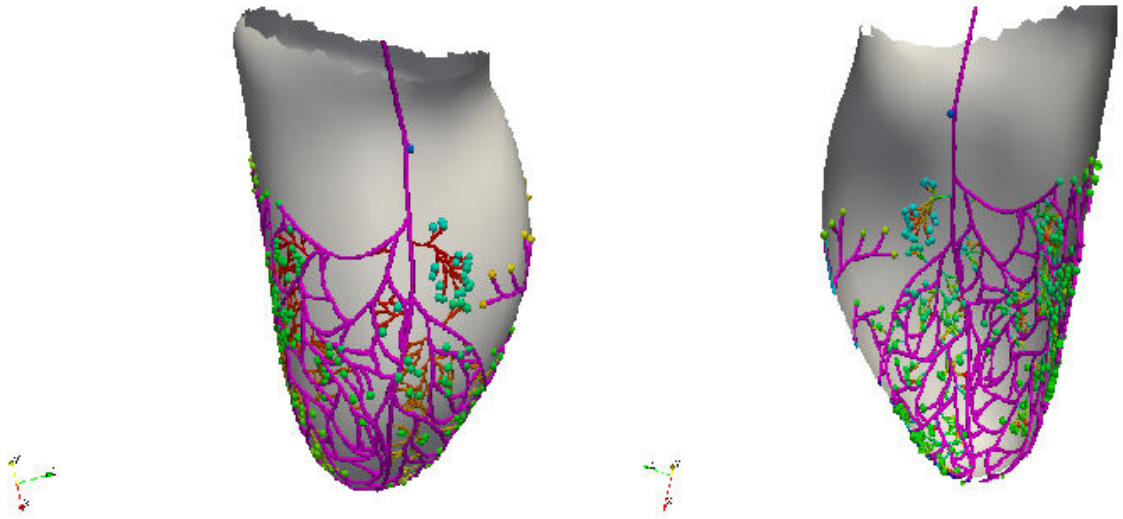

Figura 72. Septal visto desde la derecha (1er) y desde la izquierda (2º)

Como podemos ver (fig. 72), tenemos una rama principal que baja hasta la base y de la cual parten gran variedad de ramas hasta alcanzar las paredes de la estructura.

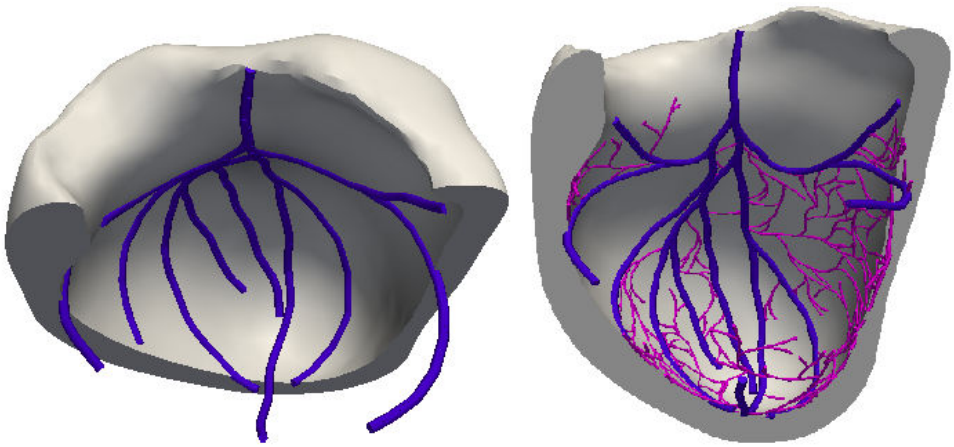

Figura 73. Primera y segunda fase de la generación

La figura (fig. 73) de la izquierda es el resultado de la primera generación del árbol. Como podemos ver al principio se generan pocas ramas. A la derecha tenemos el resultado de la segunda fase con las ramas secundarias ya formadas y partiendo de las principales.

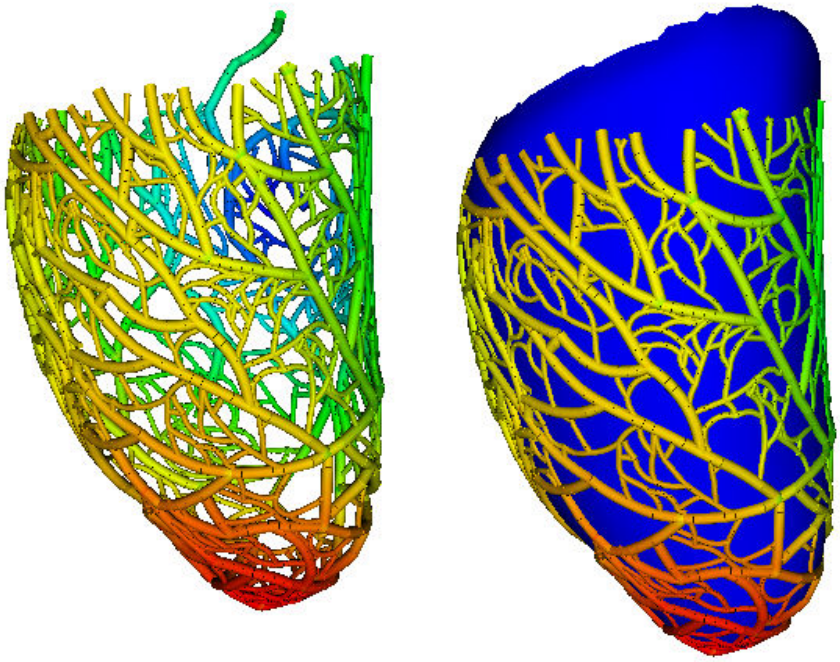

Figura 74. Red de *Purkinje* totalmente generada

En la figura anterior (fig. 74) podemos ya vislumbrar como queda nuetra red de *Purkinje* final y como se estructura alrededor de las paredes del corazón.

Por último vamos a comprobar si se parecen o no a los resultados de las imágenes de histología.

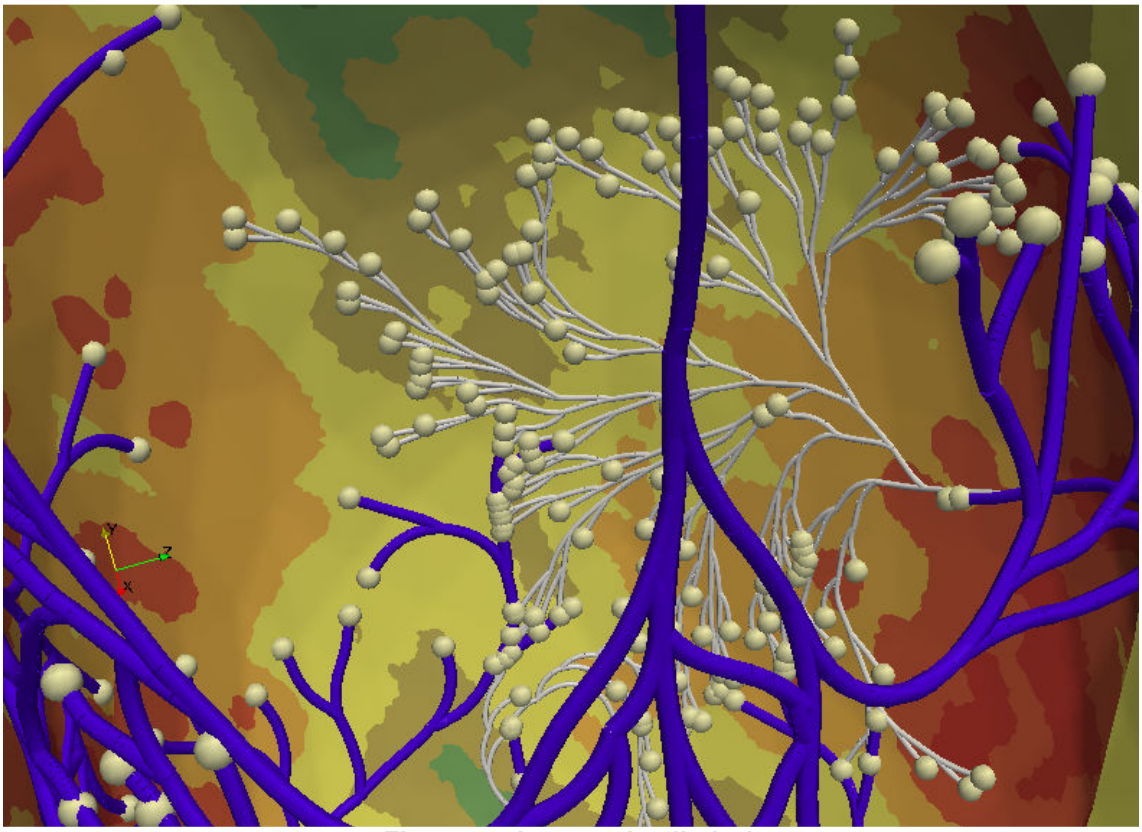

Figura 75. Imagen detallada 1

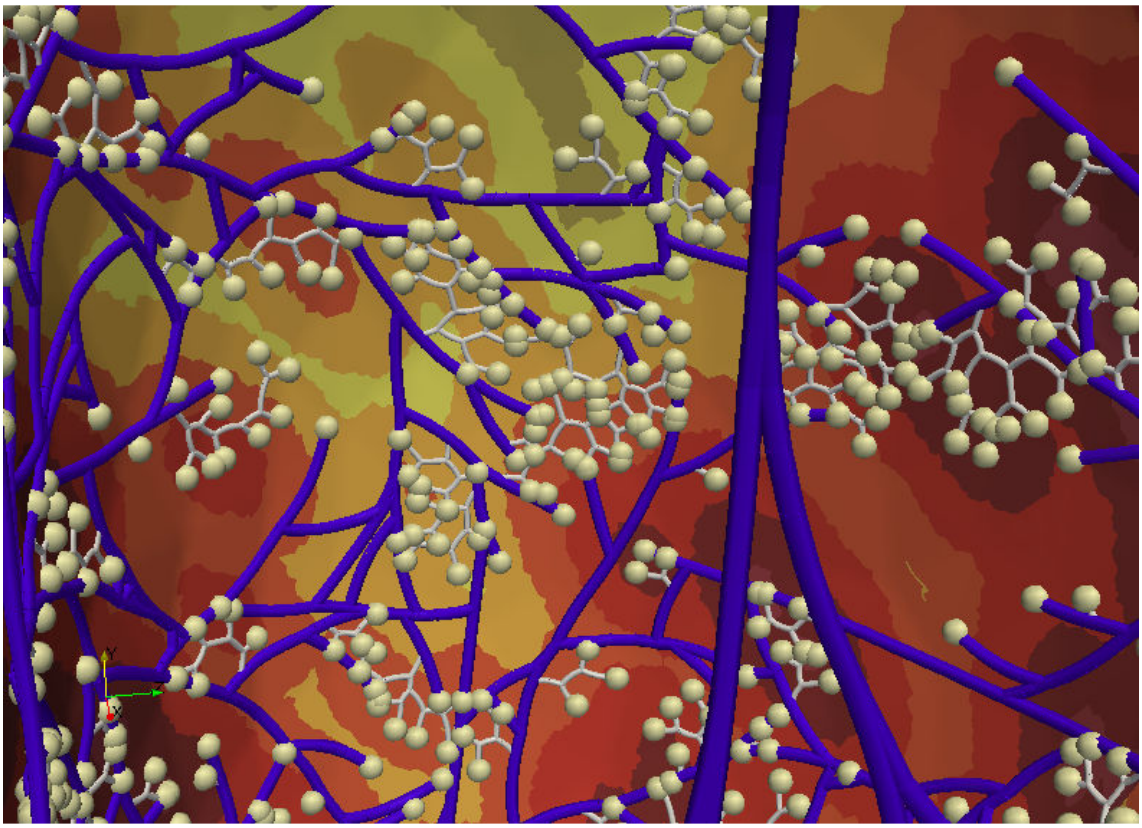

Figura 76. Imagen detallada 2

Si volviésemos atrás en el documento hasta la figura 4, podríamos asegurar que resultan muy parecidas. Por lo que a falta de la certificación médica, podemos asegurar que nuestro software da como resultado una red de *Purkinje* totalmente correcta y parecida a la que existe en la realidad.

A continuación vamos a realizar una serie de gráficas a titulo observatorio para ver en qué zonas están presentes. Para ello hemos dividido el área del corazón en 17 segmentos como se ve en la siguiente figura (fig. 77)

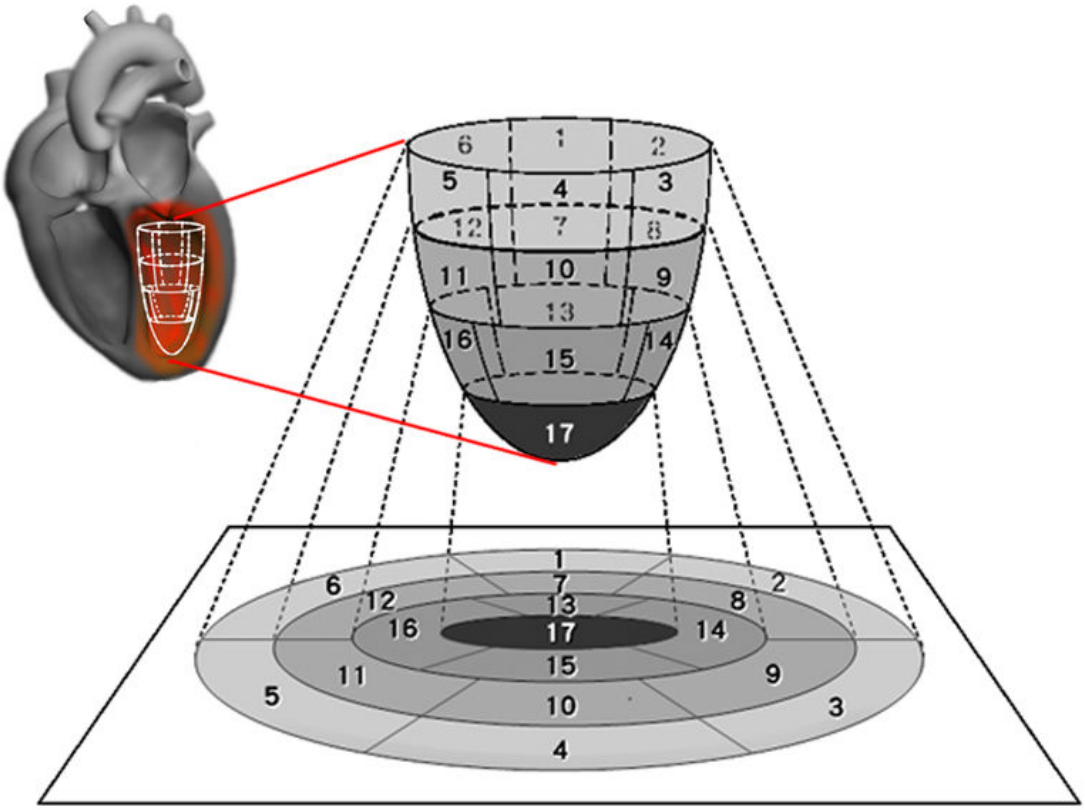

Figura 77. División en 17 segmentos

Como se ve cada región es independiente de la otra puesto que la geométrica del corazón es un elipsoide. Por ello es mejor tratar cada región de forma individual.

Para la realización de los gráficos hemos agrupado los resultados obtenidos durante la generación de los árboles de *Purkinje* de forma que obtenemos 5 grupos. Los hemos reunido según la longitud de las nuevas ramas (0.5, 1, 2, 3, 4 mm). De esta manera nos facilita la comparación entre la forma original y alternativa.

Hemos realizado dos bloques. Con el primero queremos estudiar la influencia de la cantidad de ramas que se generan tanto en segunda como en tercera fase. El segundo comprobaremos la distancia media entre los nodos del árbol según regiones.

Para ello contamos con cinco gráficas por bloque que vamos a poner y posteriormente comentar. Al final del documento, en el anexo V expondremos las tablas que hemos realizado así como una serie de gráficos que nos muestran el número total de terminales que se crean según la opción de generación realizada.

Como influyen el número de ramas en su distribución:

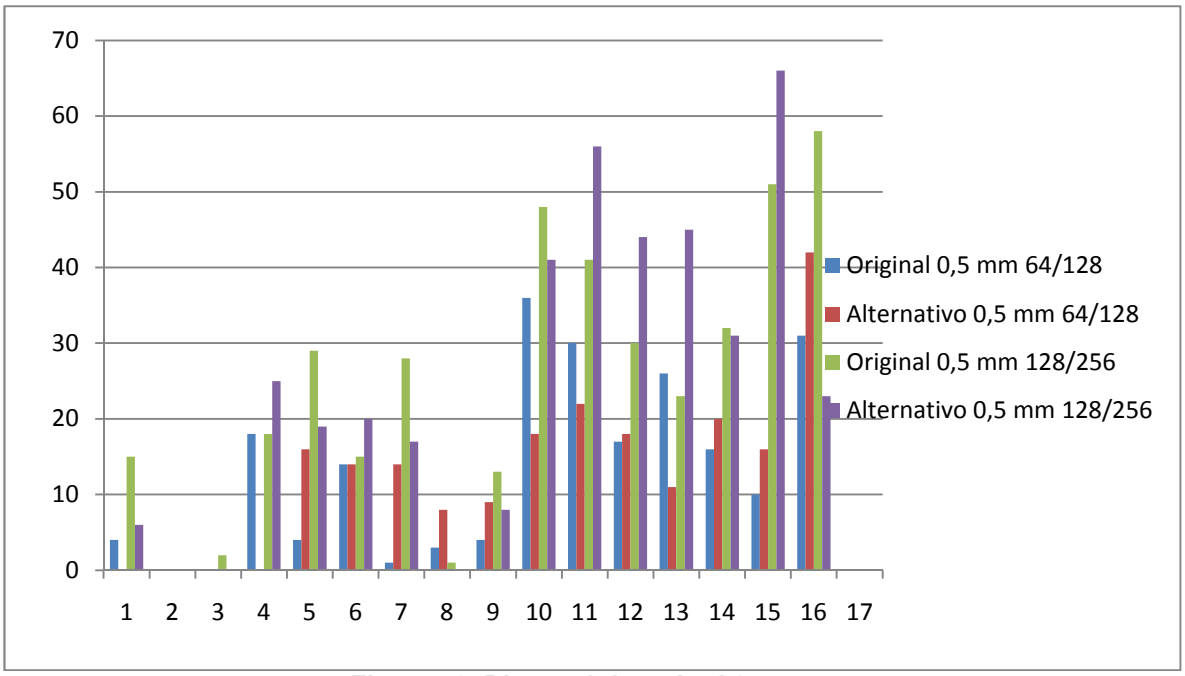

Figura 78. Bloque 1. Longitud 0.5 mm

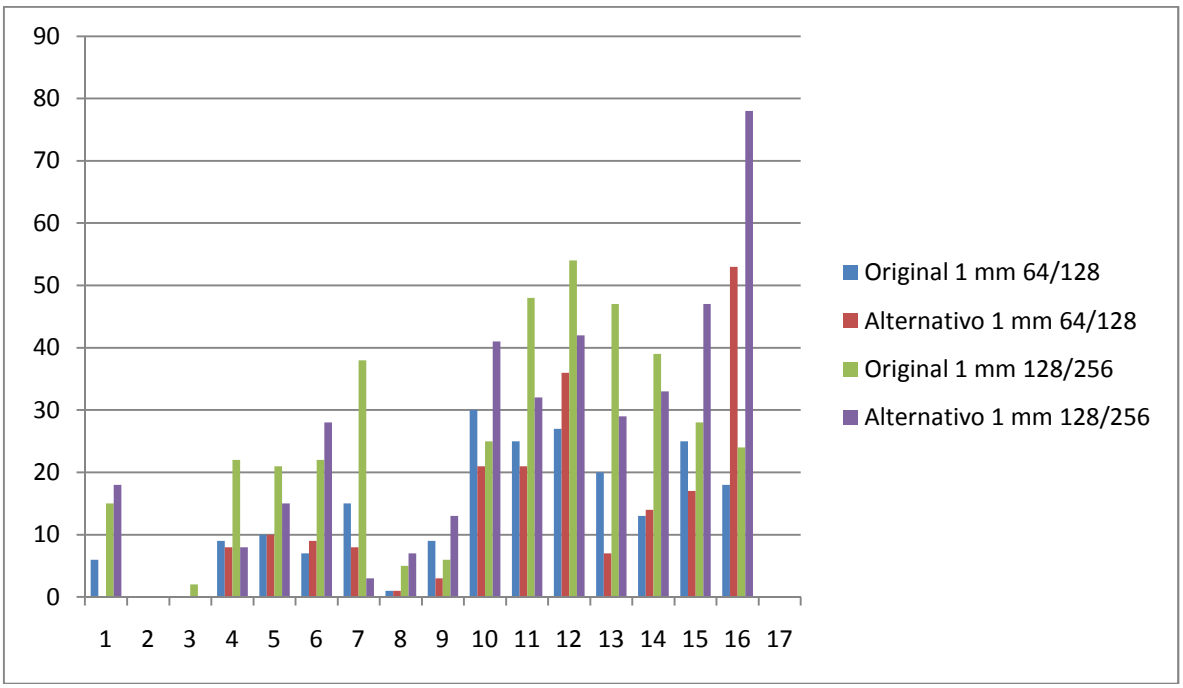

Figura 79. Bloque 1. Longitud 1 mm

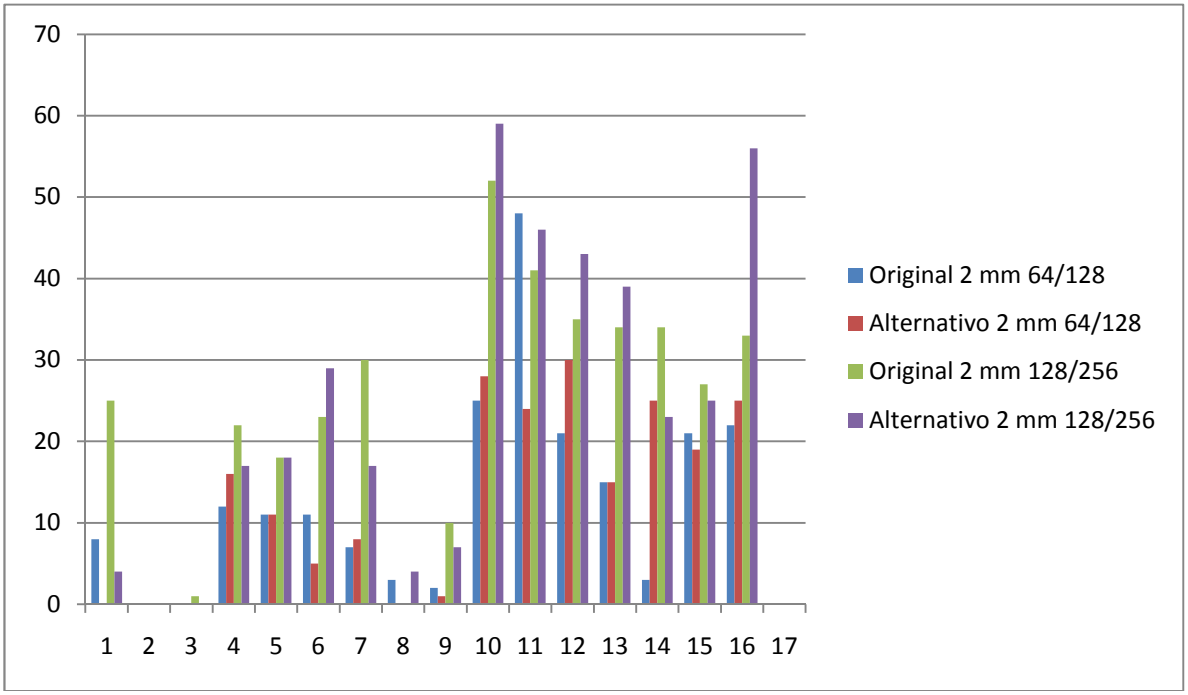

Figura 80. Bloque 1. Longitud 2 mm

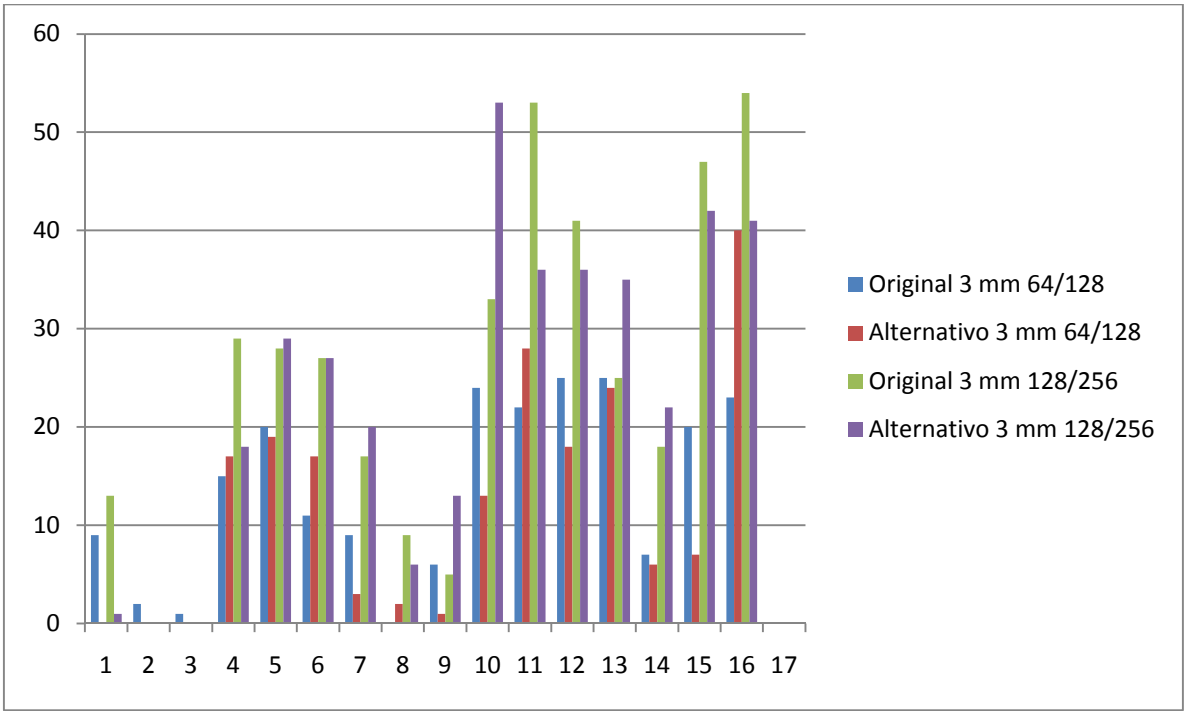

Figura 81. Bloque 1. Longitud 3 mm

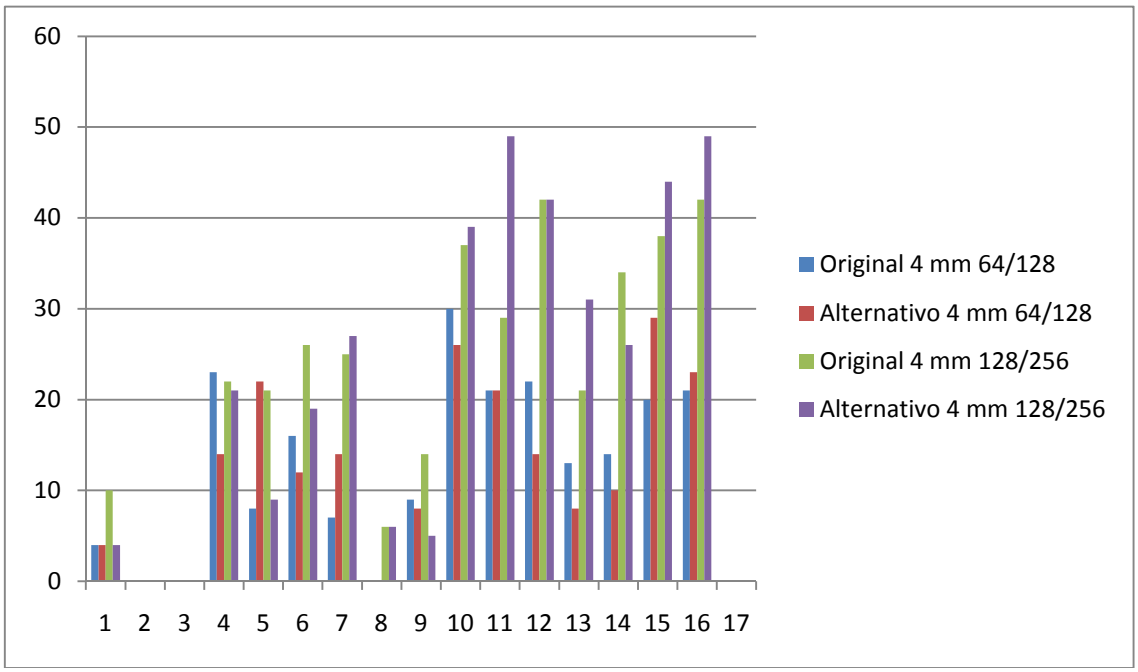

Figura 82. Bloque 1. Longitud 4 mm

Influencia del método de generación en la distancia media:

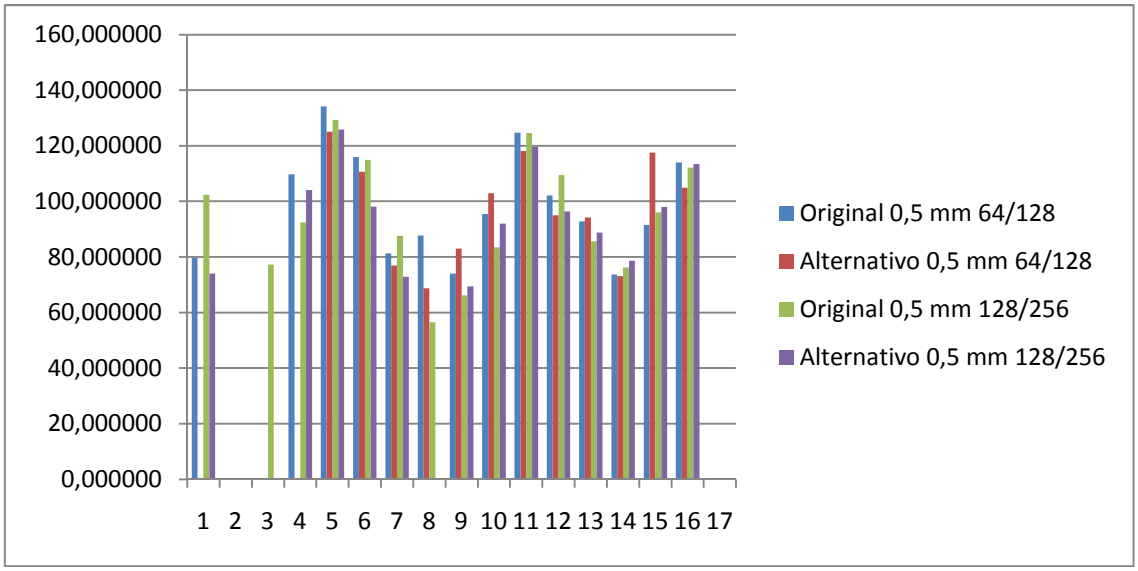

Figura 83. Bloque2. Longitud 0.5 mm

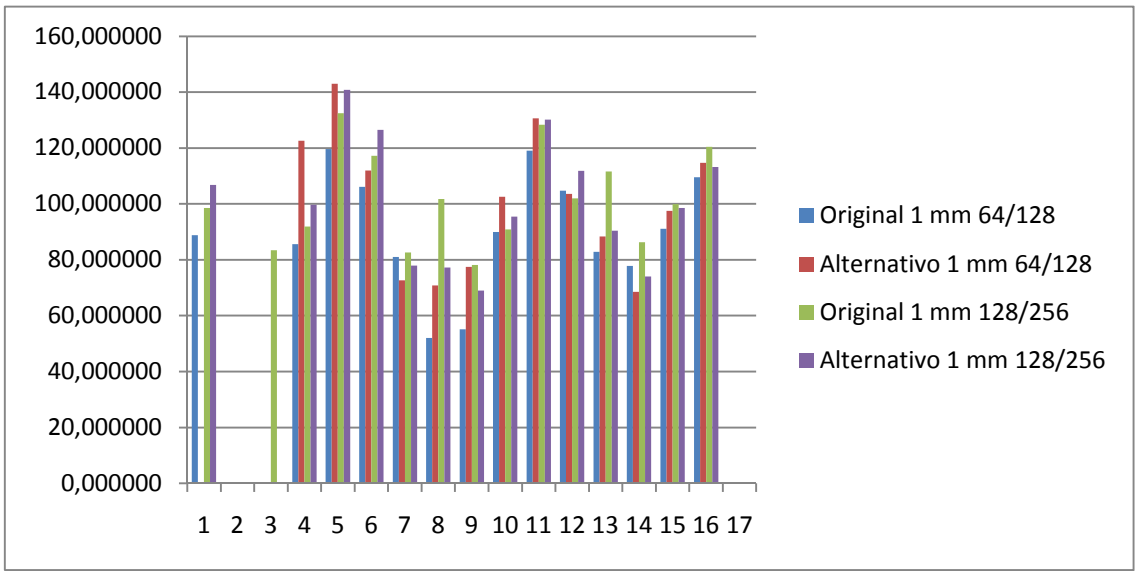

Figura 84. Bloque 2. Longitud 1 mm

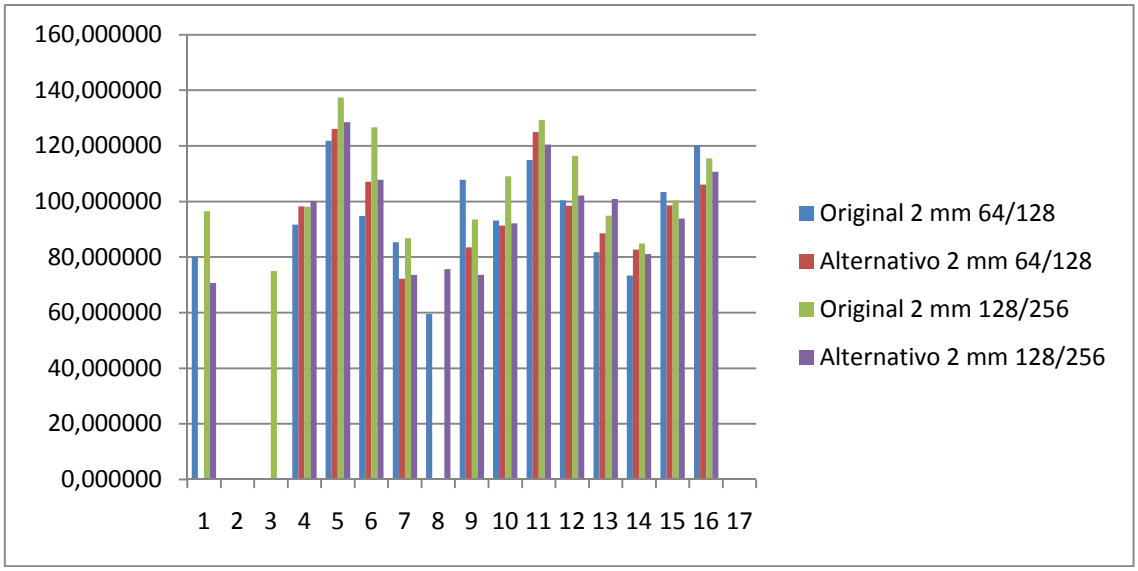

Figura 85. Bloque 2. Longitud 2 mm

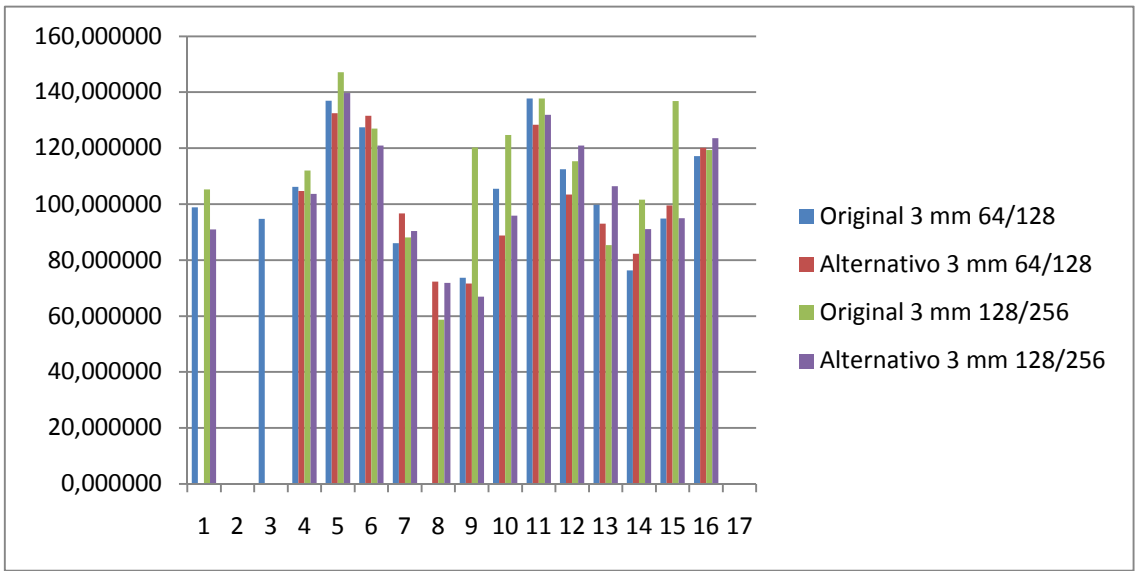

Figura 86. Bloque 2. Longitud 3 mm

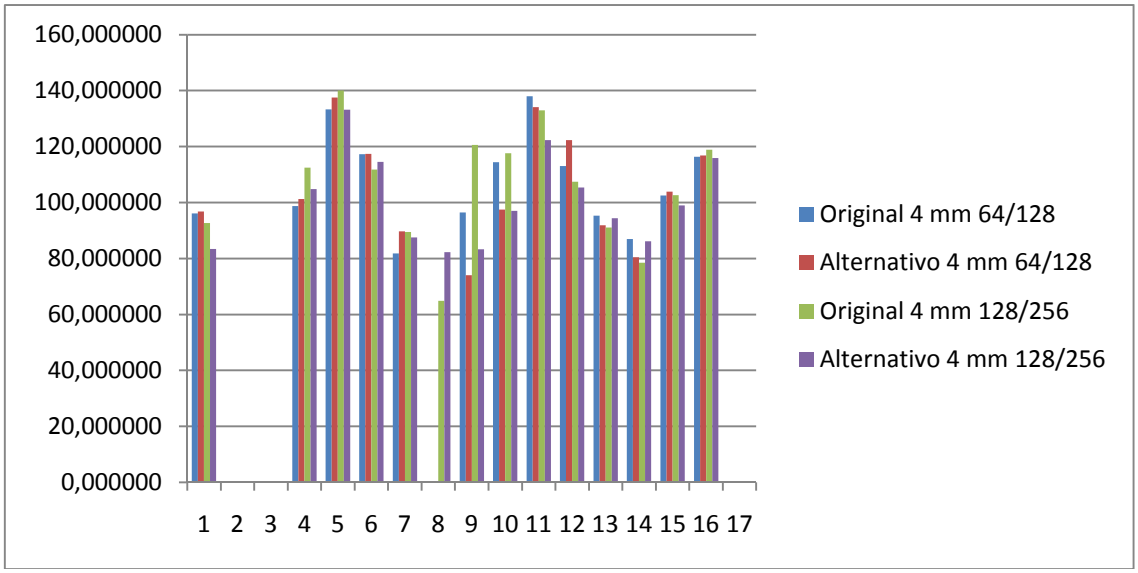

Figura 87. Bloque 2. Longitud 4 mm

Como podemos ver en ambos bloques de experimentos, hay regiones que no tienen ninguna rama y eso es debido a que por requisitos médicos, hay zonas que no van a contener ninguna rama. Es muy difícil comentar las gráficas puesto que por sí solas ofrecen información muy parecida y además al ser una generación aleatoria, no podemos asegurar la influencia que haya podido tener la generación alternativa sobre el crecimiento y posición de los nodos terminales. La idea de poder usar el método alternativo era para que cambiase la secuencia de activación, es decir, el área al que llegaba primero la señal eléctrica. Por tanto, no se esperan cambios en número de ramas, ni terminales, ni en la estructura si lo comparas con él original.

7- Conclusiones y trabajo futuro

Una vez realizadas las pruebas pertinentes, podemos asegurar que nuestro proyecto cumple con creces los requisitos demandados por el cliente. Podemos generar una red de *Purkinje* de forma completa y caracterizada por los parámetros que le insertemos. Dicha red puede ser generada de forma original o alternativa dependiendo de los intereses que tengamos. Nuestra aplicación queda totalmente integrada dentro del *framework* denominado GIMIAS. Gracias a esto conseguimos que sea lo más accesible posible tanto a nivel de utilización (mediante una interfaz de usuario acorde a las ya existente en GIMIAS) y a nivel de distribución puesto que el *framework* GIMIAS es usado a nivel internacional.

La implementación ha sido posible mediante la inclusión en escena de una librería *ThirdParty* denominada VTK. Nuestro código conjunto a dicha librería permite procesar y generar nuestro árbol. Gracias a las características que tiene, podemos tratar correctamente los elementos geométricos de nuestra malla dando unos resultados muy satisfactorios.

Toda la generación de la estructura 3D ha sido realizada empleando un PC portátil lo que demuestra como actualmente y gracias a la gran evolución que ha ido teniendo lugar la informática, las posibilidades que ofrece son casi infinitas. Las simulaciones por computador son el futuro de las investigaciones ya que nos permiten realizar los mismos experimentos obteniendo resultados más que satisfactorios minimizando los recursos necesarios para ello.

Hemos sido capaces de utilizar este gran avance para mejorar e incluso solventar las barreras que limitaban las investigaciones en ciertos campos científicos (en nuestro caso el campo sanitario). Con estas simulaciones obtendremos un elevado número de datos que aportarán un mayor conocimiento de la estructura del corazón encargada de transportar la electricidad necesaria para crear las contracciones musculares. Aumentaremos nuestro conocimientos sobre un órgano anteriormente casi inaccesible lo que podría dar lugar a mejoras a nivel físicas de las personas y tratamiento de enfermedades cardiovasculares.

Como hemos comentado, los resultados obtenidos son muy parecidos a las imágenes histológicas que disponíamos. Pero eso no conlleva que nuestro software sea correcto a nivel biológico. Para ello un doctor debería supervisar nuestro software para que sea validado y digamos “certificado”. Es importante puesto que daría mucha más fiabilidad a las investigaciones realizadas mediante nuestra aplicación. Esto conllevaría un aumento notorio de su uso.

Habría que realizar una serie de estudios que permitiesen determinar las diferencias de nuestro software con respecto a los otros ya existentes. Habría que ver sus puntos fuertes como los flojos para mejorarlos en un futuro.

Hay que tener en cuenta que este campo está en constante evolución por lo que en un futuro, a medio largo plazo después de la validación médica, nuestro software puede quedar obsoleto. Por otro lado, la informática también evoluciona

a pasos agigantados provocando que hoy nuestro software, con sus limitaciones, se ejecute de forma rápida pero que en un futuro podamos incluir muchas más opciones y se generen el doble de rápido. Se desempeñarían acciones de modificación/mejora para insertar los nuevos descubrimientos que se vayan consiguiendo. Cabe mencionar que actualmente se está trabajando mucho con el tema del 3D y quién sabe si algún día seremos capaces de obtener un corazón totalmente tridimensional que fuese proyectado por un elemento y con el que pudiésemos interactuar y realizar pruebas permitiendo unos resultados más visuales.

Referencias

[1] - F. Sachse, *Computational Cardiology. Modeling of anatomy, electrophysiology, and mechanics*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer, 2004.

[2 - A. J. Camm, T. F. Lscher, and P. Serruys, Eds., *The ESC Textbook of Cardiovascular Medicine*. Oxford, UK: Wiley-Blackwell Publishing, 2006.

[3] - Organización Mundial de la Salud
http://www.who.int/es/

[4] - Harvey Gould, Jan Tobochnik, and Wolfgang Christian, *An Introduction to Computer Simulation Methods*, July 31, 2005.

[5] - Przemyslaw Prusinkiewicz, Mitra Shirmohammadi, and Faramarz Samavati: L-systems in geometric modeling. *Proceedings of the Twelfth Annual Worskshop on Descriptional Complexity of Formal Systems*, pp. 3-12, 2010.

[6] – L-system
http://en.wikipedia.org/wiki/L-system

[7] – Entorno de prototipado medico GIMIAS
http://www.gimias.org/

[8] – Evolución médica
http://en.wikipedia.org/wiki/Timeline_of_medicine_and_medical_technology

[9] – Evolución graficos por computador
http://sophia.javeriana.edu.co/~ochavarr/computer_graphics_history/historia/
http://www.zauberklang.ch/timeline.php

[10] - Aoki, Masanori; Okamoto, Yoshiwo; Musha, Toshimitsu; Harumi, Ken-Ichi, *Three-Dimensional Simulation of the Ventricular Depolarization and Repolarization Processes and Body Surface Potentials: Nornal Heart and Bundle Branch Block*, Department of Applied Electronics, Tokyo Institute of Technology, June 1987.

[11] - S Abboud, O Berenfeld and D Sadeh, *Simulation of high-resolution QRS complex using a ventricular model with a fractal conduction system. Effects of ischemia on high-frequency QRS potentials,* Biomedical Engineering Program, Faculty of Engineering, Tel Aviv University, Israel.

[12] - Pollard, A.E.; Barr, R.C., *Computer simulations of activation in an anatomically based model of the human ventricular conduction system,* Nora Eccles Harrison Cardiovascular Res. & Training Inst., Utah Univ., Salt Lake City, UT, USA.

[13] - Omer Berenfeld, , José Jalife, Purkinje-Muscle Reentry as a Mechanism of Polymorphic Ventricular Arrhythmias in a 3-Dimensional Model of the Ventricles

[14] - K. Simelius, J. Nenonen, R. Hren and B.M. Horacek, *Anisotropic Propagation Model of Ventricular Myocardium.*

[15] - Vigmond EJ, Clements C., *Construction of a computer model to investigate sawtooth effects in the Purkinje system*.

[16] - Lindenmayer A, *Mathematical models for cellular interactions in development*, I & II. J. Theor Biol 18: 280–315, 1968.

[17] – CardioSolve
http://cardiosolv.com/

[18] – Chaste
http://www.comlab.ox.ac.uk/chaste/index.html

[19] – Mimics
http://www.materialise.com/mis

[20] – STSF
http://www.imaging.robarts.ca/petergrp/simulation-platform

[21] – DICOM
http://dicom.nema.org/

[22] – VTK
http://www.vtk.org/

[23] – Esfuerzo basado en Casos de Uso
http://pisuerga.inf.ubu.es/rcobos/anis/estimacion-del-esfuerzo-basada-en-casos-de-usos.pdf

Anexo I – VTK

I.1- Los orígenes de VTK

Originalmente el código de VTK era parte de un libro titulado "*The Visualization Toolkit An Object-Oriented Approach to 3D Graphics".* Will Schroeder, Ken Martin y Bill Lorensen, tres investigadores gráficos, escribieron el libro y lo acompañaron del software en Diciembre de 1993 y con su permiso legal se empleó en GE Corporate R & D. Los autores escribieron el libro motivados por colaborar con otros investigadores y desarrollar un open-*framework* para la creación de una aplicación líder en visualización y generación de gráficos. VTK surgió de la experiencia de los tres autores en GE, particularmente con el sistema gráfico orientado a objetos LYMB. También fue influenciado por el sistema de visualización VISAGE desarrolladla por Schroeder, el sistema informático de animación orientado a objeto denominado "the Clockworks" desarrollado en RPI y el gran éxito del libro de modelado orientado a objetos "Object-Oriented Modeling and Design" con Bill Schroeder como co-autor.

Después de que el núcleo de VTK fuese escrito, los usuarios y desarrolladores de todo el mundo comenzaron a mejorarlo y a aplicar el sistema a los problemas del mundo real. En concreto "GE Medical Systems" y otros negocios de GE contribuyeron generosamente al sistema. Algunos investigadores como el Dr. Rheinghans Penny comenzaron a dar clases con el libro. En los últimos años, Sandia National Labs han sido partidarios y cooperadores en el desarrollo con especial énfasis en la adición de visualización de información en VTK. Para apoyar lo que se estaba convirtiendo en un gran activo, Ken y Will junto con Lisa Avila, Charles Law y Bill Hoffman dejaron GE Research y fundaron Kitware, Inc en 1998. Desde entonces, cientos de desarrolladores adicionales han creado lo que ahora es el sistema de visualización de primera clase en el mundo.

I.2- El modelo gráfico

El modelo gráfico (fig. 20) captura las características esenciales del sistema gráfico 3D en una forma que pueda resultar sencilla de entender. En el modelo hay presentes nueve objetos básicos:

1. *Render Master:* coordina los métodos independientes del dispositivo y crea las ventanas renderizadas.
2. *Ventanas renderizadas:* administra una ventana en el dispositivo de ejecución. Uno o más renders se pueden dibujar en una ventana de renderización para crear una escena.
3. *Renderers:* coordina el procesado de la luz, las cámaras y los actores.
4. *Luz:* ilumina a los actores en la escena.
5. *Cámara:* define la posición de la vista, el punto focal y otras características de la cámara.
6. Actor: un objeto renderizado en una escena. Los actores están definidos en términos de mapeado, propiedades y una transformación de objetos.

7. Propiedades: representan los atributos de renderización de un actor incluyendo el color, la luz, el mapeado de texturas, el estilo del dibujado y el estilo del sombreado.
8. Mapeado: representa la definición geométrica de un actor y mapea el objeto a través de una tabla de búsquedas. Más de un actor puede referir al mismo mapeado
9. Transformación: un objeto consistente en una matriz de transformación 4x4 y métodos para modificarla. Especifica la posición y orientación de los actores, cámaras y luz.

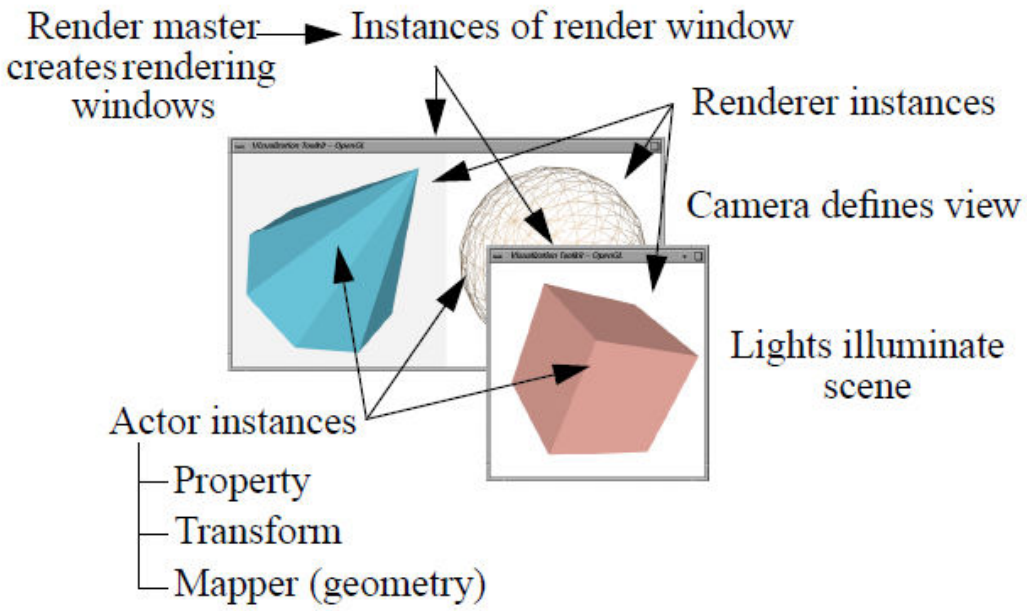

Figura 88. Modelo gráfico

En esta figura (fig. 88) podemos ver un poco resumido cada uno de los puntos vistos anteriormente. Cabe hacer notar que los objetos derivados de esta clase base, están permitidos para extender la funcionalidad del *toolkit*.

Para conseguir la portabilidad del sistema, se ha creado un concepto de objeto-dispositivo. Estos objetos son clases derivadas de superclases abstractas, o extienden la funcionalidad de clases gráficas. Luces, cámaras, propiedades y actores son ejemplos de objetos genéricos que tienen contrapartes dependientes del dispositivo. Cuando estos objetos se crean, los objetos dependientes del dispositivo se crean automáticamente y se añaden a la librería gráfica. Todo esto se realiza de forma transparente para el usuario y asegura que todas las aplicaciones son completamente independientes. Solo los objetos dependientes del dispositivo necesitan ser codificados.

I.3- VTK por dentro

Como bien sabemos, los lenguajes de programación pueden ser de dos tipos: compilados o interpretados. Los lenguajes compilados tienen un mayor rendimiento que los interpretados pero estos ofrecen una mayor flexibilidad. Las aplicaciones interpretadas pueden ser construidas más rápido que las compiladas gracias a la eliminación del ciclo de enlace/compilación. Además las aplicaciones interpretadas suelen estar escritas a un nivel más alto que las compiladas. Con esto podemos decir que el código compacto es más rápido para escribir y depurar pero se necesita del lenguaje compilado si lo que queremos es crear aplicaciones con un alto rendimiento visual. Los compilados también ofrecen un bajo nivel de acceso a los recursos de sistema. La idea es que VTK tenga lo mejor de ambos métodos (fig. 89).

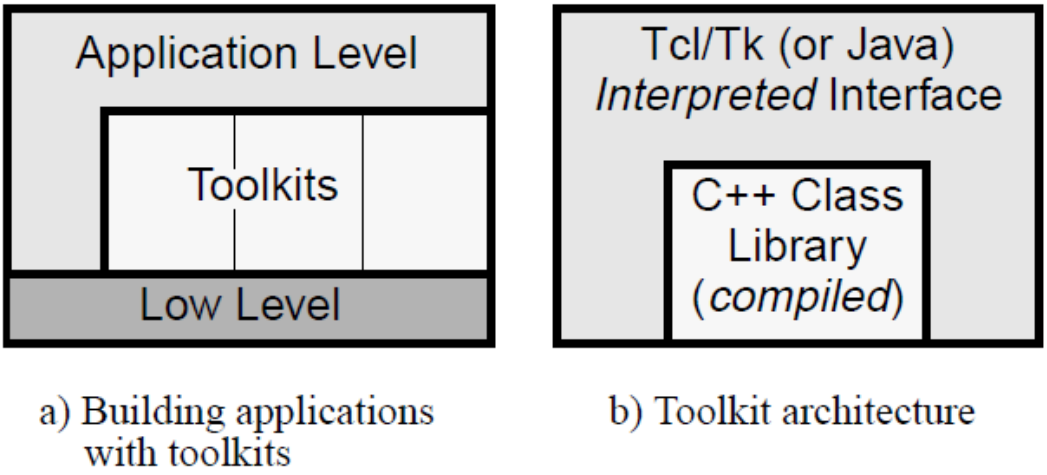

Figura 89. Arquitectura del sistema

Con esta idea de partida, se ha creado un *toolkit* cuyos objetos del núcleo computacional se han creado con un lenguaje compilado y las aplicaciones de un nivel superior con uno interpretado. Dispone una frontera entre ambas partes bastante notable por lo que se ha conseguido que el núcleo pueda separarse fácilmente de una aplicación y usarse en otra diferente. De esta manera conseguimos un alto grado de portabilidad. Por otro lado dentro de esta gran portabilidad que le confiere el núcleo, el *toolkit* se puede usar en cualquier SO gracias a que el propio núcleo está creado lo más genérico posible.

La elección del lenguaje compilado para crear el *toolkit* VTK fue predeterminado por la necesidad de un núcleo orientado a objetos. Por ello se eligió C++. Además de esto, C++ ofrece otras características importantes. Se puede emplear con una amplia selección de herramientas de desarrollo y compiladores. Es un lenguaje de tipo. Esta característica se usa en el *toolkit* para que la conectividad entre los objetos de proceso y datos se realice correctamente en la red de visualización.

En cuanto a la parte del *toolkit* del lenguaje interpretado, para mantener la independencia con el núcleo, se barajaron varias posibilidades (Tcl, Python y Perl). Finalmente se eligió Tcl por su popularidad y por su conjunto de *widget*s Tcl basados en Tk. Con la elección de Tcl/Tk se consiguió un entorno de desarrollo

potente que permitía construir interfaces de usuario complejas. Finalmente se decidió por el uso de Java pero se deja abierto para futuras posibilidades.

I.4- Multiprocesamiento

Permite la computación en paralelo (fig. 90) para reducir los tiempos de generación de los gráficos 3D.

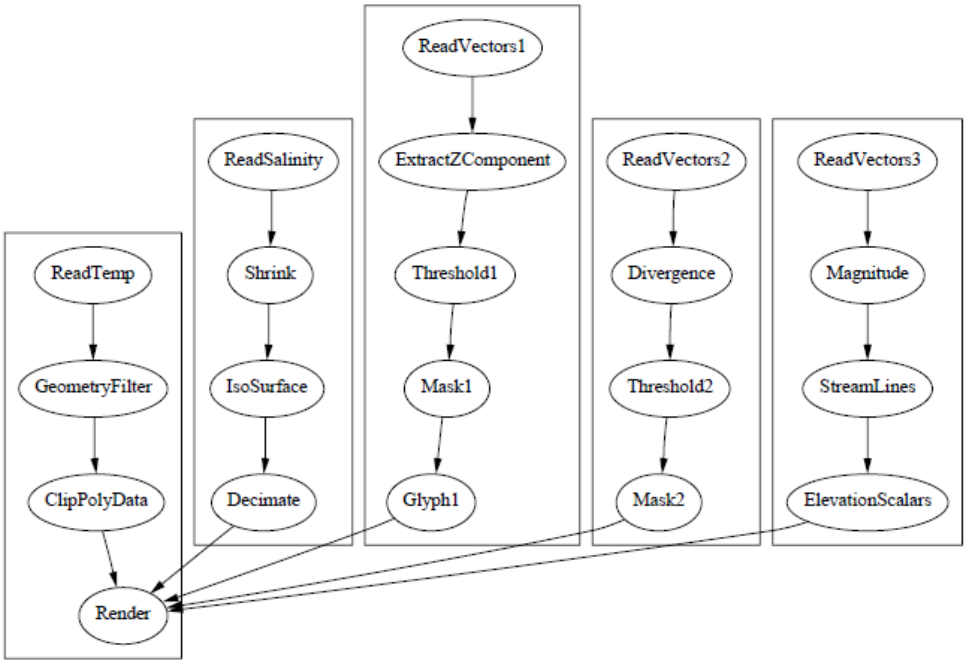

Figura 90. Ejemplo de resolución de tareas en paralelo

Con esto podríamos resolver tareas diferentes a la vez gracias a la independencia de los módulos y posteriormente juntar estos resultados para obtener el resultado final. Además de esto permite el tratamiento de datos en paralelo (fig. 91).

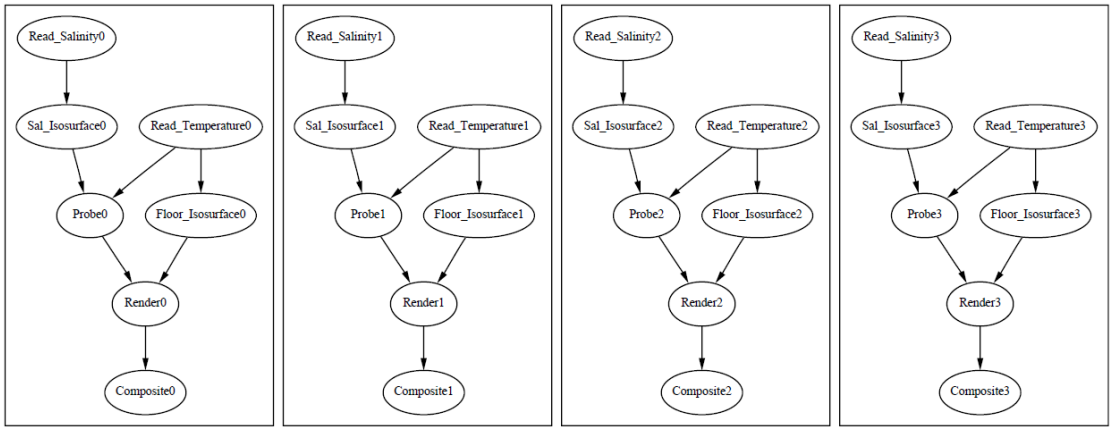

Figura 91. Tratamiento de datos en paralelo

I.5- El modelo de visualización

El modelo VTK se basa en el paradigma de flujo de datos. En este paradigma, los módulos están conectados entre sí en una red. Los módulos van realizando operaciones aritméticas a medida que los datos los van atravesando. La ejecución de esta red de visualización (fig. 92) está controlada en respuesta a la demanda de los datos o en respuesta a una entrada del usuario. El atractivo de este modelo es que es flexible y puede adaptarse rápidamente a diferentes tipos de datos o nuevos algoritmos implementados.

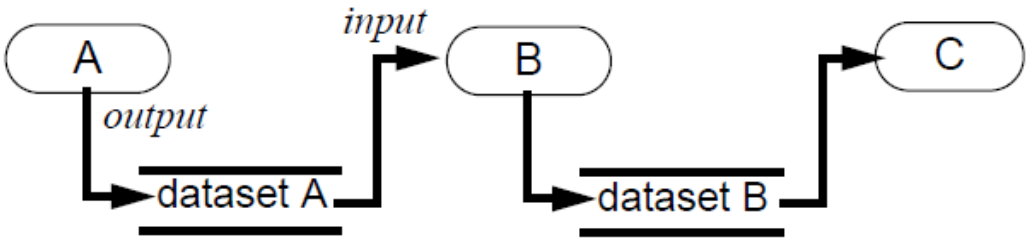

Figura 92. Modelo de visualización. Los objetos de proceso A, B y C generan diferentes salidas y entradas de uno o más objetos de datos

La visualización está compuesta de dos tipos de objetos diferentes: los objetos de proceso y los objetos de datos. Los objetos de proceso son módulos o porciones de algoritmos de la red de visualización. Los objetos de datos (también llamados *dataset*) representan y permiten operaciones en los datos que fluyen a través de la red.

Los objetos de proceso pueden ser de tres tipos: fuentes, filtros y mapeados. Los objetos fuentes inician la red y generan una o más salidas de *dataset*. Los filtros requieren una o más entradas y generan una o más salidas. Los mapeados, los cuales necesitan una o más entradas, terminan la red.

En VTK inicialmente seleccionamos cinco tipos de datos como pueden verse en la siguiente figura (fig. 94).

I.6- Manejo de la memoria

Una preocupación importante en la aplicación de visualización en forma de flujo de datos es la cantidad de memoria consumida. El *toolkit* aborda esta cuestión mediante la aplicación de un esquema contador de referencias y permitiendo al usuario adaptar la red para favorecer el cálculo o la memoria.

El contador de referencias permite a los objetos de proceso compartir objetos de datos o porciones de objetos de datos. Como ilustra la mitad superior de la figura, si porciones de datos sin modificar pasan a través de la red, los datos pueden ser referenciados por otros objetos sin necesidad de que se dupliquen. La llave a este acercamiento es el mantener el seguimiento del número de objetos

que referencian a un objeto de contador de referencias. Cuando el contador de referencias llega a cero, el objeto referenciado se elimina el mismo.

En algunas aplicaciones los recursos de memoria son escasos, mientras que en otros el costo para calcular la salida de un filtro o la red es alto. El conjunto *toolkit* proporciona las comodidades para acomodarse a estas necesidades (fig. 93). Las redes pueden ser adaptadas para favorecer la preservación de la memoria con el coste de cálculo adicional o el cálculo puede ser favorecido al coste de un requerimiento adicional de la memoria o una combinación de ambos. Estas capacidades se han implementado mediante indicadores para controlar la eliminación automática de los datos de salida cuando la red se ejecuta. Por ejemplo en la mitad inferior de la figura, si el indicador está activo y los recursos de memoria se ven favorecidos, después de que finalice la ejecución del objeto de proceso B, señala las entradas del objeto de proceso A para que eliminen sus datos de salida. Favorecer la memoria significa que A siempre se va a volver a ejecutar si alguna parte de la red que depende de A necesita volver a ejecutarse. Por el contra, si el cálculo se ve favorecido, A no se re-ejecutara hasta que sus datos de entrada sean modificados. En su lugar, A mantiene sus salidas en memoria, y puede proveer esto sin que sea necesario el cálculo de B.

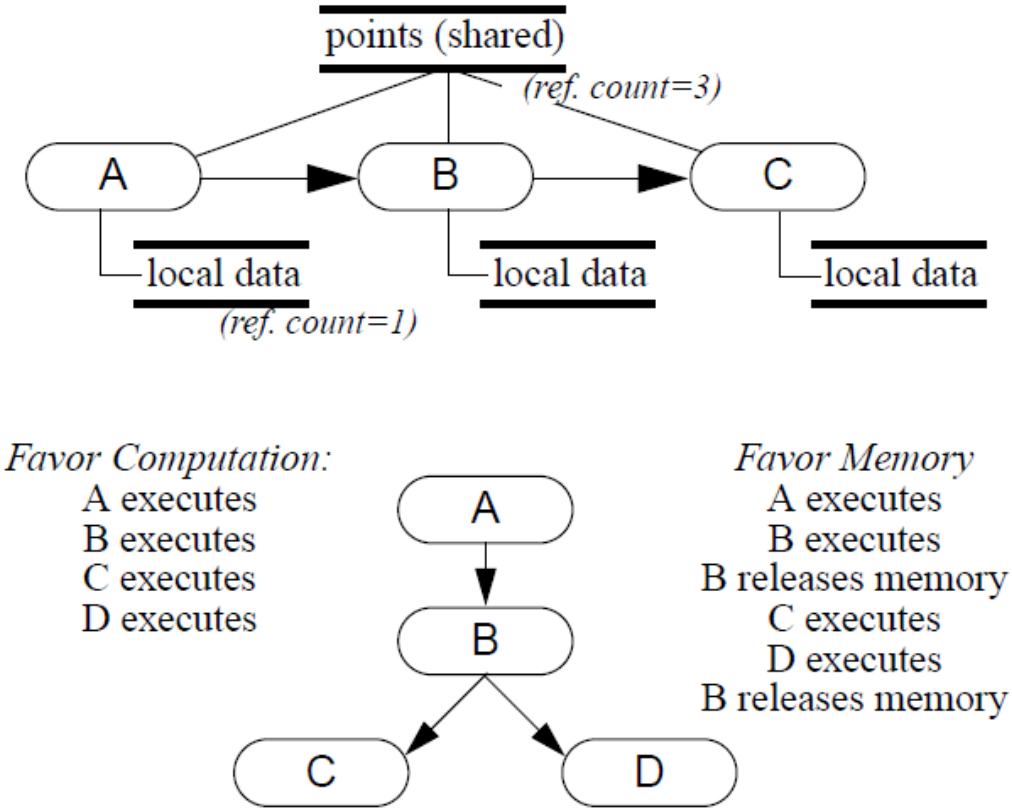

Figura 93. Manejo de la memoria. Arriba el contador de referencias. Compensación memoria/cálculo

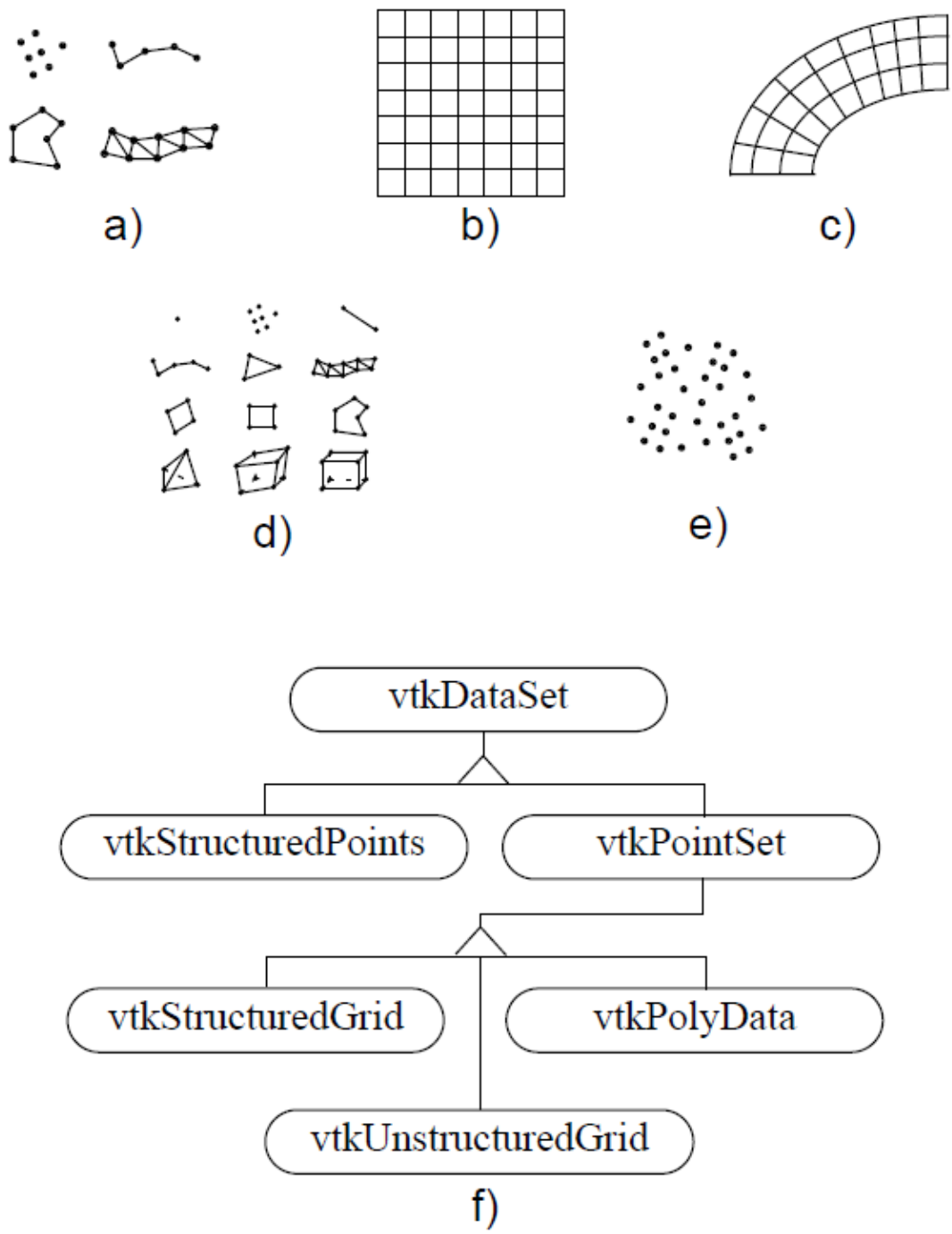

Figura 94. Tipos de datos. a) datos poligonales, b) puntos estructurados, c) malla estructurada, d) malla desestructurada, e) puntos desestructurados y f) diagrama de objetos

Según como está indicado en la figura, una interfaz abstracta a los datos está especificada por un objeto de tipo *dataset*. Las subclases del *dataset* incluyen datos poligonales, puntos estructurados, y mallas estructuradas/desestructuradas. Además era necesario definir otro objeto abstracto, el punto de ajuste, el cual es una superclase de objetos con puntos de coordenadas explícitos representados. El quinto tipo de dato, puntos

desestructurados, no está representado por si solo puesto que se puede conseguir por uno o más de los otros tipos.

Una característica importante que tiene el modelo de datos de VTK son las celdas. Un *dataset* está constituido por una o más celdas. Cada celda está considerada como una primitiva atómica de visualización. Las celdas representan las relaciones topológicas entre los puntos que conforman el *dataset*. La función principal de estas celdas es la de la interpolación de datos localmente o calcular derivados de los datos.

I.7- Topología de la red y ejecución

La construcción de la red de visualización es un proceso de conexión de los objetos de procesos y de datos. Los mayores problemas que podemos encontrar son el asegurarse que las entradas a un objeto de procesos tienen el tipo correcto y que los bucles sin terminar en la red estén correctamente tratados. Una vez que la red está correctamente creada, se necesita un mecanismo de actualización como datos de entrada o cambios de los parámetros de los objetos.

Los bucles en la red ocurren cuando la entrada a un objeto B es la salida de un objeto D, donde D depende de la salida de B. Esto lo podemos ver en el apartado b de la figura. Aunque ambas situaciones no son comunes, se pueden usar para obtener ciertas ventajas. Por ejemplo la integración numérica de un conjunto de puntos a través de un campo del vector puede ser simulado por la red vista en el apartado b de la figura. Aquí, el movimiento del punto denominado P está controla por el vector de valores P, y cuando P se mueve, el vector de valores se vuelve a muestrear para las nuevas posiciones de P. El proceso se repite mientras se va moviendo P a través del *dataset*.

En VTK, los bucles infinitos se evitan mediante un *flag* interno. Lo objetos de proceso establecen este *flag* cuando se empiezan a ejecutar. Este *flag* previene que ocurran las recursiones infinitas en la red devolviendo el objeto de proceso. Por lo tanto, los bucles están permitidos pero sólo se ejecutan una vez por ejecución de la red.

La ejecución de la red está basada en un esquema implícito. En este esquema cada objeto de proceso mantiene un *timer* de modificación interno y un *timer* de ejecución. Luego cuando una salida de un objeto de proceso A se solicita, A compara su *timer* de modificación interna y el *timer* de modificación de la entrada contra el *timer* de ejecución. Si A ha sido modificado, o su entrada se ha modificado más recientemente que su último registro del *timer* de computación, entonces A se volverá a ejecutar. Hay dos partes para actualizar la red utilizando este esquema implícito: un pase de actualización que compara los *timers* de modificación y ejecución y un pase de ejecución en el cual los objetos de proceso tienen que volver a ser ejecutados para que se actualicen.

Anexo II – Esfuerzo por Puntos de Casos de Uso

II.1- Definición

El cálculo del esfuerzo por Puntos de caso de uso es un método de estimación de esfuerzo para proyectos de software, a partir de sus casos de uso. Fue desarrollado por Gustav Karner en 1993, basándose en el método de punto de función, y supervisado por Ivar Jacobson. Ha sido analizado posteriormente en otros estudios, como la tesis de Kirsten Ribu (Universidad de Oslo) en 2001.

El método utiliza los actores y casos de uso relevados para calcular el esfuerzo que significará desarrollarlos. A los casos de uso se les asigna una complejidad basada en transacciones, entendidas como una interacción entre el usuario y el sistema, mientras que a los actores se les asigna una complejidad basada en su tipo, es decir, si son interfaces con usuarios u otros sistemas. También se utilizan factores de entorno y de complejidad técnica para ajustar el resultado.

El método de punto de casos de uso consta de cuatro etapas, en las que se desarrollan los siguientes cálculos:

1. Factor de peso de los actores sin ajustar (UAW)
2. Factor de peso de los casos de uso sin ajustar (UUCW)
3. Puntos de caso de uso ajustados (UCP)
4. Esfuerzo horas-hombre

II.2- Esfuerzo y duración

Al inicio de un proyecto de software, cuando apenas se conocen los casos de uso y sus actores asociados, se puede proyectar una breve descripción de cada caso de uso, en el cual se describe de forma breve la funcionalidad que éste debe brindar.

El UUCP son los puntos de casos de uso sin ajustar, esto nos puede servir para tener una idea un poco más precisa de la dificultad de los casos de uso e interfaces, tomando en cuenta los pesos de los actores (UAW) y los pesos de los casos de uso (UUCW). UUCP = UAW + UUCW

Estas siglas significan:

- UUCP: Puntos de casos de uso sin ajustar.
- UAW: Factor de peso de los actores sin ajustar.
- UUCW: Factor de peso de los casos de uso sin ajustar.

Aplicando el análisis de puntos de función a estos casos de uso, se puede obtener una estimación trivial del tamaño y a partir de ella una estimación del esfuerzo.

Una vez calculado UUCP, procederíamos a calcular los Puntos de Casos de Uso ajustados (UCP). Para esto se utilizan las siglas UCP y se obtiene al multiplicar el UUCP el TCF y el EF quedando la operación de la siguiente forma:

UCP = UUCP x TCF x EF

Estas siglas significan:

- UCP: Puntos de casos de uso ajustados.
- UUCP: Puntos de casos de uso sin ajustar.
- TCF: Factores técnicos (13 factores diferentes).
- EF: Factores ambientales (8 factores diferentes).

Una vez obtenido el valor de UCP, hay que modificarlo para poder expresarlo en Esfuerzo (hora-hombre). Para ello aplicamos un filtro según los valores obtenidos en la tabla de Factores ambientales. Hay que contar la cantidad de factores ambientales del E1 al E6 que estén por debajo de 3 y de E7 y E8 por encima de 3. Se suman ambos resultados. Si obtenemos un valor inferior o igual a 2, el Factor de conversión (CF) es 20. Si es inferior o igual a 4 CF vale 28. En cambio si es mayor o igual de 5, CF es 36.

Anexo III – csPkjTime, csSurfaces y csStructUtils

III.1- csPkjTime

Es la clase encargada de calcular el tiempo que tarda en generarse nuestra estructura. Se inicia cuando iniciamos la generación de la primera fase y concluye cuando termina de generarse la tercera.

Nos sirve para hacernos una idea de lo que le cuesta al algoritmo, medido en tiempo, en generar la solución siempre dependiendo de los parámetros de entrada.

Contiene diez métodos, de los cuales uno es el constructor y otro el destructor de la clase. El resto vamos a exponerlos a continuación.

"**computeActivTimesInTerminals**": computa la distancia mínima entre el punto inicial a cada uno de los terminales y almacena la información en un objeto de tipo vtkPolyData.

"**getMaxBranchDistance**": computa la distancia mínima entre el punto inicial a cada una de las ramas.

"**getNumberOfTerminals**": devuelve el número de terminales.

"**GetPd**": devuelve el PD con los terminales y sus tiempos de activación.

"**readStimulesParametersFile**": lee el archivo Stimuli con los parámetros.

"**createXMLwithActivTimes**": almacena en un archivo xml las coordenadas de cada punto junto con su tiempo de activación.

"**terminalsData17Seg**": computa el número de terminales y la distancia media mínima al punto inicial en cada una de las 17 regiones.

"**computeDistances**": calcula el camino de longitud mínima desde el punto inicial de la red a cada uno de los terminales (se asume que las ramas tienen la dirección correcta).

"**getDataFromPdTree**": lee la información sobre cada una de las ramas de la estructura.

"**setTerminalsAndLs**": almacena en un vector las coordenadas de los puntos y su camino de longitud mínima con respecto al punto inicial de la red.

III.2- csSurfaces

Esta clase es la encargada del procesado previo de la superficie de la estructura. La llamaremos siempre antes de empezar a construir nuestra red de *Purkinje* puesto que antes de empezar hay que realizar ciertas acciones sobre la estructura como definir las regiones y etiquetarlas. Contiene 31 métodos (contando el constructor y el destructor de clase).

"**SplitSurface**": divide la superficie de la malla en endo-epi, ventrículo izquierdo/derecho y determina el tipo de malla (LV, RV, biV),

"**SetInitPkjPt**": encuentra las coordenadas correspondientes al punto inicial del árbol.

"**SetExtraPkjPt**": dados los tres puntos (centro, ápex y punto de inicio) encuentra un cuarto sobre el plano central y con la dirección perpendicular a la recta entre el centro y el punto inicial.

"**GetScalarName**": devuelve el ScalarName que contiene la información de las regiones (sus escalares), etc…

"**GetScalarSimuName**": devuelve el ScalarName para las simulaciones.

"**SetScalarNames**": establece el nombre para ScalarName y para ScalarSimuName. Por defecto toman los valores "regionID" y "17SegRegionID" respectivamente.

"**GetSurfaceMesh**": devuelve el objeto de tipo vtkPolyData.

"**SetSurfaceMesh**": establece el objeto vtkPolyData.

"**GetMeshType**": devuelve el valor del tipo de malla.

"**NewLabels**": devuelve cierto si se han actualizado las etiquetas de la malla y falso en caso contrario.

"**Has*Purkinje***": devuelve cierto su la superficie contiene el sistema de *Purkinje*.

"**GetNrosurfs**": devuelve el numero de superficie (2 para ventricular, 4 para biventricular y 0 para error).

"**GetSurfaceNormals**": devuelve un punto de acceso a las normales de una de las sub partes de la superficie.

"**GetSurfaceLocator**": devuelve un punto de acceso a las celdas de localización de una de las sub partes de la superficie.

"**GetMitralCentroid**": devuelve las coordenadas del centroide mitral.

"**GetApex**": devuelve las coordenadas del ápex.

"**GetExtraPkjPt**": devuelve las coordenadas de los puntos de *Purkinje* extras.

"**GetInitPkjPt**": devuelve las coordenadas del punto de inicio de la red.

"**GetRVanularCentre**": devuelve el centro del ventrículo derecho.

"**GetRVapex**": devuelve las coordenadas del ápex del ventrículo derecho.

"**GetRVExtraPkjPt**": devuelve las coordenadas de RVextraPkj.

"**LoadSurfaceMesh**": lee el archivo vtk que contiene la superficie.

"**SetConstants**": establece los valores de las etiquetas para identificar regiones, acorde con las nuevas o viejas definiciones.

"**processSurface**": procesa toda la información en la superficie.

"**GetLVaxisDirection**": devuelve la dirección normalizada de los ejes principales del ventrículo izquierdo.

"**GetLVaxisLength**": calcula y devuelve la longitud de los ejes principales.

"**CalculateLVaxisDirectionAndLength**": calcula la longitud y dirección normalizada de los ejes principales.

"**SetFix17SegmentsRegions**": establece un escalar con las etiquetas para las 17 regiones de la superficie. Las regiones las elije mediante la información de un fichero.

"**Calculate17SegmentsRegions**": establece un escalar con las etiquetas para las 17 regiones de la superficie. Calcula las regiones.

III.3- csStructUtils

Genera un conjunto de métodos necesarios para realizar ciertos cálculos durante la generación. Son métodos de apoyo y van desde calcular procesos matemáticos hasta recuperar valores o mostrar mensajes.

"**ExitMessage**": sale del programa mediante un mensaje.

"**ExitMessage2**": sale del programa mediante dos mensajes.

"**change_extn**": remplaza la extensión del fichero con una nueva. Si no tiene extensión simplemente añade la nueva.

"**Data**": devuelve un string con los datos que hay entre el primer espacio en blanco que encuentra hasta el siguiente.

"**LoadBulkNodes**": lee los nodos generados con TetGen y crea el objeto de tipo vtkPolyData.

"**ConvertBulkMeshUg2PdNodes**": lee los nodos desde vtk UnstructuredGrid y crea el objeto de tipo vtkPolyData.

"**SaveBulkNodes_Carp**": almacena los nodos de acuerdo con CARP, números reales específicos en micrómetros y sin una columna extra.

"**GetFurthestPoint**": obtiene el punto más a dentro de shapePt desde un punto de referencia.

"**GetClosestPoint**": obtiene el punto más cercano de shapePt desde un punto de referencia.

"**GetPolyDataRegion**": obtiene la región de la malla de tipo vtkPolyData la cual su nombre tiene valor entre 'start' y 'end'.

"**GetPolyDataConnectedSubpart**": obtiene una región umbral de una forma. Aplica un filtro de conectividad a la malla de tipo vtkPolyData para dividir la malla en diferentes regiones. Devuelve el número asociado a cada sub parte y lo almacena en 'shapeOut'.

"**GetNumberPolyDataConnectedSubparts**": aplica un filtro de conectividad a la malla de tipo vtkPolyData para dividir la malla en diferentes regiones conectadas. Devuelve el número de las regiones conectadas.

"**GetPolyDataConnectedSubparts**": aplica un filtro de conectividad a la malla de tipo vtkPolyData para dividirla en diferentes regiones conectadas. La salida es un objeto de tipo vtkPolyData con un escalar extra que especifica cada región.

"**GetPolyDataEdge**": obtiene el borde de una región conectada en una malla 'shapeIn'.

"**RefineCurvedLine**": devuelve una línea curva como un conjunto de puntos 'lineIn', crea un 'Cardinal Spline' usando sus nodos y lo refina para tener los nodos separados de una distancia inferior o igual a 'd'.

"**nPointsRefined**": mide la longitud de una línea curva 'lineIn' y calcula el numero de nodos necesarios para tenerlos separados por una distancia 'd'.

"**branchLength**": mide la longitud de una línea curva 'lineIn'.

"**setScalarsInRefinedBr**": establece los escalares en una rama refinada acorde a los escalares en la rama original.

"**ComputeSurfaceNormals**": calcula la normal a cada una de las celdas de la superficie (no nodos).

“**InsertPolyDataPoint**”: inserta un punto en la última posición de un vtkPolyData que consiste en las coordenadas de los nodos y líneas.

“**InsertDupPdPoint**”: inserta ‘nNp’ puntos repetidos en la línea de tipo PolyData.

“**InsertExtraPdPoint**”: inserta 2 puntos repetidos en la línea de tipo PolyData.

“**DeleteExtraPdPoint**”: elimina un punto repetido en la línea de tipo PolyData.

“**MergePointsInPolyD**”: limpia la malla de tipo vtkPolyData uniendo puntos coincidentes.

“**CreatePDFfromPtsArray**”: crea un objeto de tipo vtkPolyData usando los puntos almacenados en el vector ‘Pts’.

“**AppendPDwithPtsArray**”: añade los puntos almacenados en el vector ‘Pts’ al objeto de tipo vtkPolyData.

“**AppendPDs**”: añade meshPd2 a meshPd1.

“**AddPrevPts2PdLines**”: añade un escalar con los puntos previos a cada punto de una línea de un objeto de tipo PolyData conteniendo solo líneas.

“**AddIDs2PD**”: añade un escalar a un objeto de tipo vtkPolyData y establece el mismo valor a todos los puntos.

“**rotateVect**”: rota el vector ‘V’ y el ángulo ‘alpha’ respecto a la dirección ‘Z’.

“**quicksort**”: ordena dos vectores (juntos) en orden ascendente usando el Quicksort con particiones en 3 direcciones.

“**exch**”: intercambia el valor de dos variables entre ellas.

“**linesDistance3D**”: calcula la distancia entre dos líneas en 3D.

“**gaussRand**”: generador de números aleatorios normales.

“**box_muller**”: otro generador de números aleatorios.

“**PDFInverseGaussianDistrib**”: calcula la función de densidad de probabilidad.

“**inverseGaussianDistrib**”: genera variables aleatorias con una distribución gausiana inversa.

Anexo IV – Tablas de los experimentos

IV.1- Bloque de pruebas 1: Incremento del número de ramas

Región	Original 0,5 mm 64/128	Alternativo 0,5 mm 64/128	Original 0,5 mm 128/256	Alternativo 0,5 mm 128/256
1	4	0	15	6
2	0	0	0	0
3	0	0	2	0
4	18	0	18	25
5	4	16	29	19
6	14	14	15	20
7	1	14	28	17
8	3	8	1	0
9	4	9	13	8
10	36	18	48	41
11	30	22	41	56
12	17	18	30	44
13	26	11	23	45
14	16	20	32	31
15	10	16	51	66
16	31	42	58	23
17	0	0	0	0
TOTAL	214	208	404	401

Tabla 15. Tabla de densidad para el bloque 1 longitud 0.5 mm

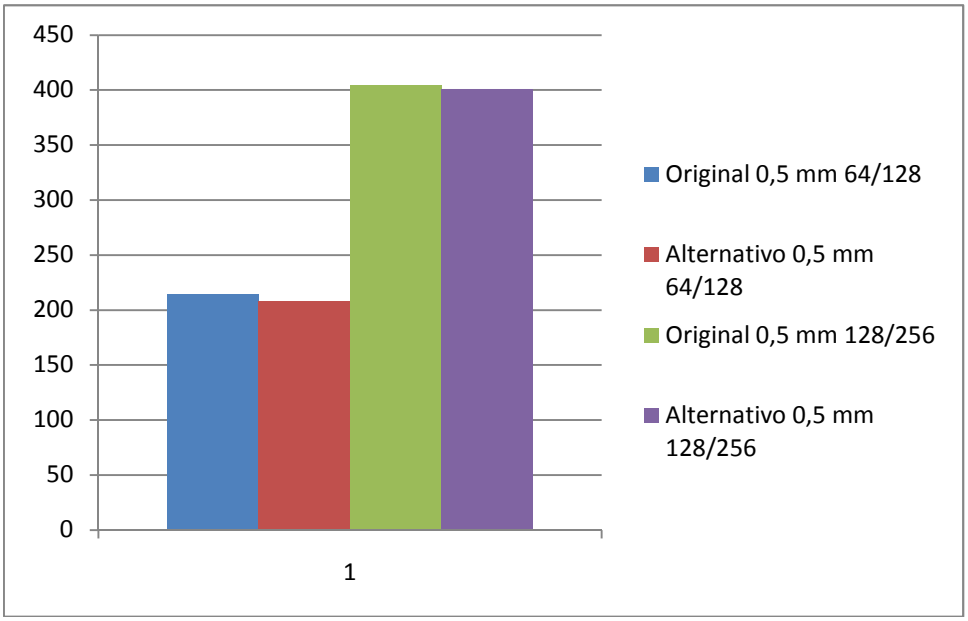

Figura 95. Comparación totales bloque 1 longitud 0.5 mm

Región	Original 1 mm 64/128	Alternativo 1 mm 64/128	Original 1 mm 128/256	Alternativo 1 mm 128/256
1	6	0	15	18
2	0	0	0	0
3	0	0	2	0
4	9	8	22	8
5	10	10	21	15
6	7	9	22	28
7	15	8	38	3
8	1	1	5	7
9	9	3	6	13
10	30	21	25	41
11	25	21	48	32
12	27	36	54	42
13	20	7	47	29
14	13	14	39	33
15	25	17	28	47
16	18	53	24	78
17	0	0	0	0
TOTAL	215	208	396	394

Tabla 16. Tabla de densidad para el bloque 1 longitud 1 mm

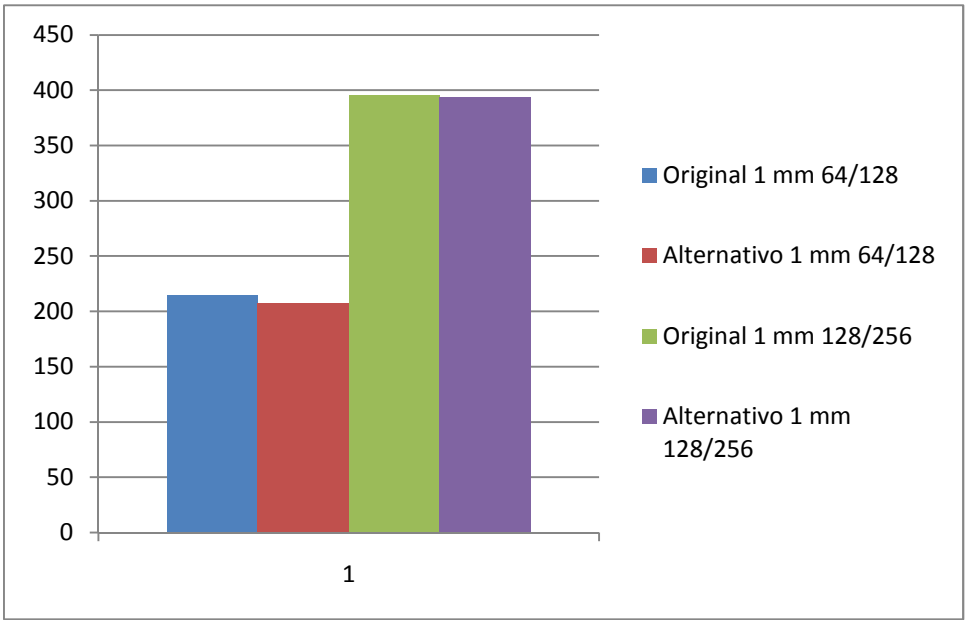

Figura 96. Comparación totales bloque 1 longitud 1 mm

Región	Original 2 mm 64/128	Alternativo 2 mm 64/128	Original 2 mm 128/256	Alternativo 2 mm 128/256
1	8	0	25	4
2	0	0	0	0
3	0	0	1	0
4	12	16	22	17
5	11	11	18	18
6	11	5	23	29
7	7	8	30	17
8	3	0	0	4
9	2	1	10	7
10	25	28	52	59
11	48	24	41	46
12	21	30	35	43
13	15	15	34	39
14	3	25	34	23
15	21	19	27	25
16	22	25	33	56
17	0	0	0	0
TOTAL	209	207	385	387

Tabla 17. Tabla de densidad para el bloque 1 longitud 2 mm

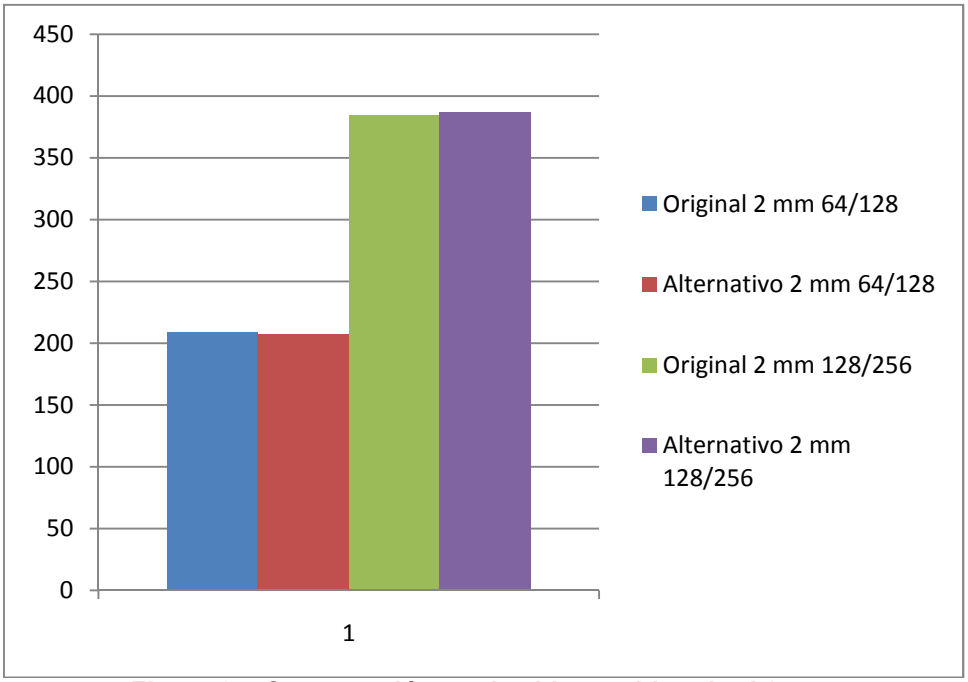

Figura 97. Comparación totales bloque 1 longitud 2 mm

Región	Original 3 mm 64/128	Alternativo 3 mm 64/128	Original 3 mm 128/256	Alternativo 3 mm 128/256
1	9	0	13	1
2	2	0	0	0
3	1	0	0	0
4	15	17	29	18
5	20	19	28	29
6	11	17	27	27
7	9	3	17	20
8	0	2	9	6
9	6	1	5	13
10	24	13	33	53
11	22	28	53	36
12	25	18	41	36
13	25	24	25	35
14	7	6	18	22
15	20	7	47	42
16	23	40	54	41
17	0	0	0	0
TOTAL	219	195	399	379

Tabla 18. Tabla de densidad para el bloque 1 longitud 3 mm

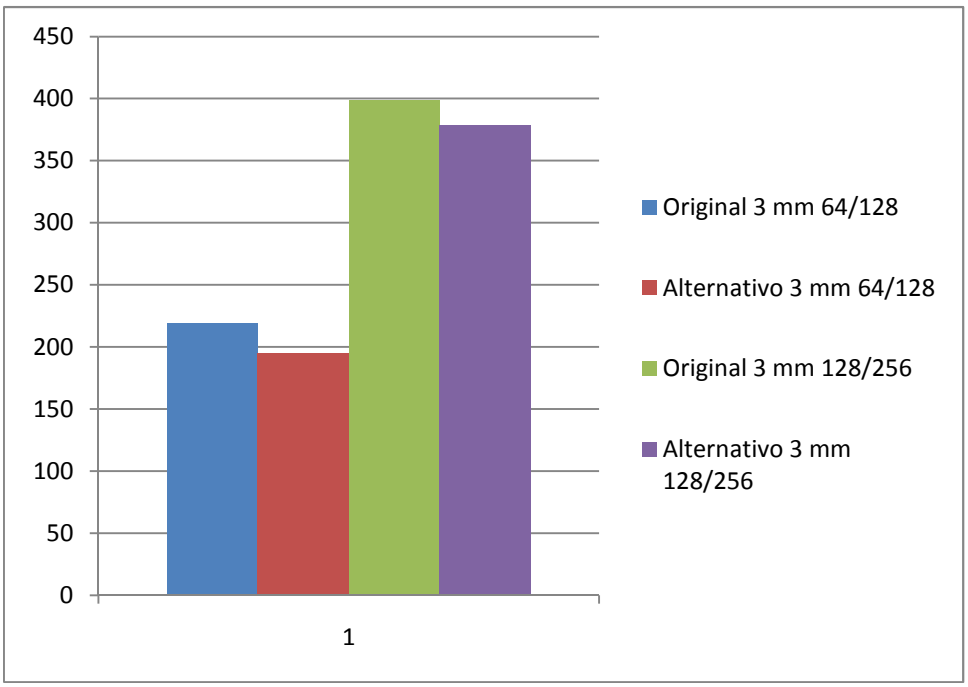

Figura 98. Comparación totales bloque 1 longitud 3 mm

Región	Original 4 m 64/128	Alternativo 4 mm 64/128	Original 4 mm 128/256	Alternativo 4 mm 128/256
1	4	4	10	4
2	0	0	0	0
3	0	0	0	0
4	23	14	22	21
5	8	22	21	9
6	16	12	26	19
7	7	14	25	27
8	0	0	6	6
9	9	8	14	5
10	30	26	37	39
11	21	21	29	49
12	22	14	42	42
13	13	8	21	31
14	14	10	34	26
15	20	29	38	44
16	21	23	42	49
17	0	0	0	0
TOTAL	208	205	367	371

Tabla 19. Tabla de densidad para el bloque 1 longitud 4 mm

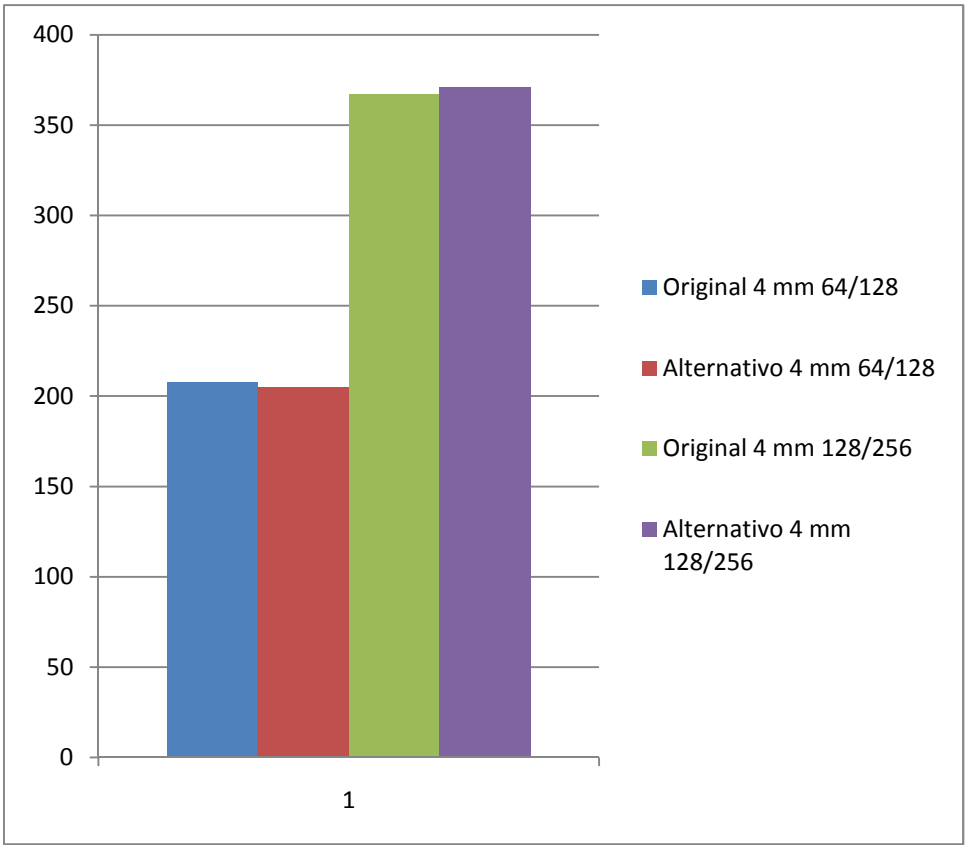

Figura 99. Comparación totales bloque 1 longitud 4 mm

IV.2- Bloque de pruebas 2: Distancia media

Región	Original 0,5 mm 64/128	Alternativo 0,5 mm 64/128	Original 0,5 mm 128/256	Alternativo 0,5 mm 128/256
1	79,610038	0,000000	102,387873	74,015794
2	0,000000	0,000000	0,000000	0,000000
3	0,000000	0,000000	77,243645	0,000000
4	109,690224	0,000000	92,447152	104,136510
5	134,179231	125,063902	129,316673	125,830897
6	115,942899	110,706042	114,879669	98,079387
7	81,245798	76,881649	87,650878	72,885579
8	87,676657	68,720411	56,532865	0,000000
9	73,969954	83,017731	66,163133	69,425528
10	95,471330	102,946723	83,483217	92,015677
11	124,651807	118,133117	124,610171	119,646809
12	102,071357	95,003211	109,494509	96,323258
13	92,846407	94,129669	85,692345	88,819007
14	73,655121	73,069597	76,242497	78,600667
15	91,524045	117,518486	96,061762	97,923013
16	114,030436	104,899921	112,105779	113,422999
17	0,000000	0,000000	0,000000	0,000000

Tabla 20. Tabla de distancia media para el bloque 2 longitud 0.5 mm

Región	Original 1 mm 64/128	Alternativo 1 mm 64/128	Original 1 mm 128/256	Alternativo 1 mm 128/256
1	88,766219	0,000000	98,554136	106,831528
2	0,000000	0,000000	0,000000	0,000000
3	0,000000	0,000000	83,423278	0,000000
4	85,628290	122,598062	91,838024	99,740566
5	119,659005	143,018390	132,419057	140,838033
6	106,099295	111,955619	117,225565	126,523951
7	80,983980	72,651006	82,593742	77,926863
8	52,082033	70,842650	101,710121	77,174823
9	55,095446	77,497108	78,145368	68,954816
10	90,001718	102,592699	90,896393	95,404949
11	119,069264	130,572519	128,271581	130,192928
12	104,754044	103,539289	101,989326	111,867125
13	82,823018	88,336299	111,618137	90,373087
14	77,782986	68,529764	86,288254	73,975847
15	91,053656	97,519423	99,946417	98,570282
16	109,539101	114,700103	120,458598	113,210636
17	0,000000	0,000000	0,000000	0,000000

Tabla 21. Tabla de distancia media para el bloque 2 longitud 1 mm

Región	Original 2 mm 64/128	Alternativo 2 mm 64/128	Original 2 mm 128/256	Alternativo 2 mm 128/256
1	80,064375	0,000000	96,492241	70,739332
2	0,000000	0,000000	0,000000	0,000000
3	0,000000	0,000000	74,986981	0,000000
4	91,616880	98,252510	98,107709	100,169694
5	121,777105	126,081311	137,366673	128,537150
6	94,745953	107,130016	126,669309	107,832618
7	85,379701	72,178505	86,881417	73,554112
8	59,528481	0,000000	0,000000	75,717050
9	107,748762	83,470054	93,554995	73,607379
10	93,208730	91,281346	109,109572	92,153148
11	114,935840	124,917570	129,326929	120,509648
12	100,423973	98,518568	116,367732	102,201766
13	81,752399	88,527628	94,940424	100,889656
14	73,379006	82,656025	84,831544	81,092052
15	103,454587	98,541183	100,440369	93,804472
16	120,174876	106,044282	115,456330	110,713943
17	0,000000	0,000000	0,000000	0,000000

Tabla 22. Tabla de distancia media para el bloque 2 longitud 2 mm

Región	Original 3 mm 64/128	Alternativo 3 mm 64/128	Original 3 mm 128/256	Alternativo 3 mm 128/256
1	98,798289	0,000000	105,295138	90,912589
2	0,000000	0,000000	0,000000	0,000000
3	94,675319	0,000000	0,000000	0,000000
4	106,138229	104,711810	111,948987	103,682212
5	136,980164	132,431510	147,075702	139,690079
6	127,419385	131,582379	126,980649	120,872396
7	86,001632	96,619712	88,080027	90,341349
8	0,000000	72,259139	58,700761	71,816474
9	73,711355	71,655674	120,294230	66,892477
10	105,518587	88,765262	124,730214	95,888925
11	137,739139	128,336462	137,802008	131,909497
12	112,497787	103,402914	115,307279	120,875234
13	99,769491	93,036036	85,338563	106,392203
14	76,262145	82,211625	101,585963	91,110934
15	94,859950	99,546663	136,807731	94,965755
16	117,199313	120,164369	119,362249	123,545995
17	0,000000	0,000000	0,000000	0,000000

Tabla 23. Tabla de distancia media para el bloque 2 longitud 3 mm

Región	Original 4 mm 64/128	Alternativo 4 mm 64/128	Original 4 mm 128/256	Alternativo 4 mm 128/256
1	96,103854	96,738562	92,711075	83,387267
2	0,000000	0,000000	0,000000	0,000000
3	0,000000	0,000000	0,000000	0,000000
4	98,785665	101,261823	112,427275	104,777873
5	133,291265	137,568513	140,080581	133,195679
6	117,220473	117,363215	111,807960	114,485120
7	81,801504	89,641473	89,436328	87,469094
8	0,000000	0,000000	64,828583	82,273317
9	96,457995	73,961737	120,543229	83,329742
10	114,402160	97,418410	117,570898	97,024943
11	137,941112	134,121303	132,921579	122,334254
12	112,984462	122,300050	107,436390	105,405619
13	95,321652	91,871079	91,042970	94,431667
14	86,915449	80,404555	78,438535	86,119218
15	102,460936	103,888759	102,618065	98,926612
16	116,342483	116,788631	118,915918	115,929855
17	0,000000	0,000000	0,000000	0,000000

Tabla 24. Tabla de distancia media para el bloque 2 longitud 4 mm