

PROYECTO FINAL DE CARRERA

SIMULADOR VIRTUAL DE
TÉCNICAS DE ENDOSCOPIA RÍGIDA
PARA FORMACIÓN SANITARIA

Abril 2014

Autor:

David Alejandro Castillo Bolado

Tutores:

Miguel Ángel Rodríguez Florido
Jorge Ballesteros Ruiz Benítez de Lugo
Agustín Trujillo Pino



UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática



Índice

Resumen del proyecto	VII
1. Objetivos	1
2. Introducción	3
2.1. Técnicas sanitarias de mínima invasión	3
2.1.1. Técnicas de endoscopia rígida	4
2.2. Simulación virtual	6
2.2.1. Simulación médica virtual	6
2.3. Entorno ESQUI	8
2.3.1. Licencia	8
2.3.2. vtkESQui	9
2.3.3. SRML	9
3. Análisis de los requisitos técnicos	11
3.1. Estudio del prototipo	11
3.2. Dispositivo periférico	13
3.3. Herramientas de desarrollo software	15
3.3.1. CMake	15
3.3.2. Doxygen	15
3.3.3. Python	15
3.3.4. GIT	16
3.3.5. VTK	16
3.3.6. vtkESQui	17
3.3.7. wxWidgets	21
4. Planificación del proyecto	23
4.1. Formación en la tecnología de desarrollo	23
4.2. Desarrollo tecnológico	24
4.3. Validación y documentación	25
5. Implementación del proyecto	27
5.1. Metodología de desarrollo	27
5.2. Código en C/C++ y clases de vtkESQui	28
5.2.1. Módulo de simulación	29

5.2.2.	Módulo de escenario	31
5.2.3.	Módulo de detección de colisiones	32
5.2.4.	Módulo de biomecánica	35
5.2.5.	Módulo de interacción	37
5.2.6.	Módulo de soporte de dispositivos periféricos	40
5.2.7.	Módulo de entrada/salida	42
5.2.8.	Casos particulares en los distintos sistemas operativos .	43
5.3.	Código en Python	45
5.3.1.	Referencias en Python	45
5.3.2.	Integración VTK-wxPython	46
5.3.3.	Visualización de colisiones	49
5.3.4.	Simulación de la visualización de la lente	52
5.3.5.	Proceso de simulación y renderizado	52
6.	Prototipo de simulador de técnicas de endoscopia rígida monocanal	55
6.1.	Funcionalidad	55
6.2.	Flujo de la aplicación	56
6.3.	Finalidad	57
7.	Manual de usuario	59
7.1.	Instalación	59
7.2.	Guía de inicio	59
7.3.	Dudas frecuentes y resolución de problemas	64
8.	Manual del desarrollador	67
8.1.	Instalación de vtkESQui	67
8.2.	Documentación del código Python	70
8.3.	Ejemplos de código de la aplicación	70
8.4.	Desarrollo futuro de vtkESQui	76
9.	Conclusiones y líneas futuras del proyecto	83
9.1.	Conclusiones	83
9.2.	Líneas futuras	84
A.	Glosario de términos	87

Resumen del proyecto

En este proyecto se trata el proceso de análisis y desarrollo llevado a cabo con el objetivo de construir un prototipo funcional de simulador virtual de endoscopia rígida monocanal orientado a la histeroscopia.

Las técnicas sanitarias mínimamente invasivas son un conjunto de técnicas sanitarias que evitan o minimizan todo lo posible los daños en el paciente. Pueden ser diagnósticas o terapéuticas y el grado de invasión en el paciente puede variar desde un pequeño punzamiento (laparoscopia) hasta un nivel de invasión nulo (ecografía). Por lo tanto, consiguen una recuperación más rápida del paciente y una reducción del dolor postoperatorio. La simulación virtual permite que el especialista pueda desarrollar sus habilidades en un entorno realista, didáctico y controlado.

Para el desarrollo del prototipo se toma como base el entorno ESQUI[1], un entorno de simulación virtual médica de código libre. Este entorno provee una librería, basada a su vez en la conocida librería gráfica VTK (Visual ToolKit)[2], cuyo propósito es poner a disposición del programador toda la algoritmia necesaria para construir una simulación médica virtual. En este proyecto, esta librería se depuró y amplió para mejorar el soporte a las técnicas de endoscopia rígida que se persiguen simular. Por otro lado se emplea el *Simball 4D*, un dispositivo de interfaz humana de la empresa *G-coder Systems*[3], para capturar la interacción del usuario emulando la morfología y dinámica de un endoscopio rígido.

Todos estos elementos se conectan con una interfaz gráfica sencilla, intuitiva y práctica soportada por *wxWidgets*[4] y utilizando Python[5] como lenguaje de scripting.

Finalmente, se analiza el prototipo resultante y se proponen una serie de líneas futuras de cara a la aplicación didáctica del mismo, tanto en relación a los objetivos conceptuales del prototipo como a los aspectos específicos del entorno ESQUI.

Capítulo 1

Objetivos

En el ámbito de la formación sanitaria, el entrenamiento de las técnicas más avanzadas es esencial de cara a disminuir los riesgos asociados para el paciente. El proceso de aprendizaje clínico habitual pasa por la realización de prácticas con modelos anatómicos, animales, cadáveres e incluso pacientes vivos. Este tipo de prácticas son actualmente un medio importante para procurar el desarrollo de las habilidades del especialista, si bien suelen adolecer de ciertos defectos.

El empleo de modelos anatómicos artificiales puede ser suficiente en los niveles de aprendizaje más básicos, pero debido a su simplicidad y artificiosidad suelen estar restringidos al entrenamiento de la coordinación básica con ejercicios de baja complejidad.

Cuando se persigue entrenar las técnicas más avanzadas, es deseable contar con entornos realistas y que preparen al especialista para reaccionar ante casos que puedan darse en la realidad. Si bien las prácticas con cadáveres cumplen con los requisitos desde el punto de vista anatómico, el hecho de emplear tejidos muertos hace que la actividad de los mismos no exista, por lo que se suele recurrir a las prácticas con animales o pacientes vivos.

La utilización de animales para el entrenamiento de técnicas sanitarias es un proceso que, además de ser costoso, requiere una gran infraestructura, pues es necesario cumplir con una serie de normativas que actualmente regulan la práctica. Por otra parte, la anatomía de un animal difiere de la de un humano y la práctica con pacientes vivos se presenta peligrosa, ya que a pesar de existir supervisión médica conlleva un riesgo elevado para el paciente involucrado.

La simulación virtual se presenta como un medio efectivo para entrenar al especialista en técnicas sanitarias mínimamente invasivas, ayudando a completar y reducir el periodo de aprendizaje experimental y clínico. No obstante, las empresas del sector suelen proveer los sistemas de simulación virtual en paquetes predefinidos para el entrenamiento de una técnica concreta y funcio-

nando con software de código privativo.

Este escenario, que motivó en su momento la aparición del entorno ESQUI, es ahora motivo de la elaboración de este proyecto, mediante el cual se pretende conseguir un perfeccionamiento de la simulación de técnicas de endoscopia rígida con herramientas de software libre y cuyos objetivos principales se tratan a continuación.

En primer lugar, se pretende **analizar el estado del arte y las necesidades de formación que existen en técnicas de endoscopia rígida**. Esto dará una visión general del contexto en el que se desarrolla este proyecto y aportará la perspectiva necesaria para su correcta aplicación docente.

A partir de este primer objetivo se persigue **desarrollar un prototipo de aplicación software para la formación en técnicas de endoscopia rígida monocanal**. Este prototipo será una primera aproximación a una aplicación docente para el entrenamiento de técnicas sanitarias orientadas a la endoscopia rígida monocanal.

Esto nos lleva al siguiente objetivo, en el que se propone **integrar el prototipo software de la aplicación con un sistema hardware de interacción hombre-máquina**. En un sistema de simulación virtual, la interacción con el usuario debe ser fiel al escenario que se simula. A tal efecto existen dispositivos periféricos que emulan el comportamiento de endoscopios rígidos y que pueden integrarse fácilmente a través de una interfaz provista por el fabricante.

Finalmente, el último objetivo exige **aplicar este desarrollo a una especialidad sanitaria específica** y diseñar el instrumental necesario para ello, así como escenarios didácticos en la técnica. Este objetivo acerca el desarrollo tecnológico del proyecto a su aplicación docente, estableciendo criterios y líneas futuras para el desarrollo de una aplicación apta para su empleo en la formación sanitaria.

Capítulo 2

Introducción

Para entender correctamente los propósitos de este proyecto, primero es necesario aclarar una serie de conceptos que ayudarán a situar el contexto correctamente.

2.1. Técnicas sanitarias de mínima invasión

La técnicas sanitarias de mínima invasión o mínimamente invasivas son aquellas que evitan o minimizan todo lo posible los daños en el paciente. Para ello, pueden servirse de distintos tipos de tecnología y de pequeñas incisiones o aperturas naturales cuando es necesario el acceso a la zona que se desea tratar.

Las técnicas menos invasivas suelen ser las diagnósticas, que inspeccionan la zona afectada del paciente sin acceder físicamente a ella. Dentro de este grupo podríamos contar la ecografía, la radiografía y la tomografía axial computerizada (aplicación avanzada de la radiografía).



Figura 2.1: Técnica de diagnóstico por ecografía.

Por otro lado se encuentran las que se practican a través de pequeñas in-

cisiones o aperturas naturales. Se utiliza para ello un sistema de asistencia visual, llamado endoscopio, que permite visualizar la zona en cuestión dentro del paciente y actuar en ella. Una técnica englobada en este grupo sería la broncoscopia, que emplea un endoscopio flexible para inspeccionar las vías respiratorias del paciente. Esta técnica suele realizarse con propósitos diagnósticos y la invasión es mínima, ya que se accede a través de las aperturas nasales o bucal del paciente.



Figura 2.2: Broncoscopia realizada por Gustav Killian en torno al año 1887.

Este tipo de técnicas, debido al menor tamaño de incisión —o ausencia de ésta—, introduce grandes mejorías de cara al paciente pues reduce el dolor postoperatorio y acelera la recuperación del mismo. Es por ello que cada vez más intervenciones se realizan con técnicas de mínima invasión, en lugar de la tradicional cirugía abierta.

2.1.1. Técnicas de endoscopia rígida

La endoscopia o endoscopía es una técnica sanitaria de mínima invasión, utilizada en medicina, que consiste en la introducción de una cámara o lente dentro de un tubo o endoscopio a través de un orificio natural, una incisión quirúrgica o una lesión para la visualización de un órgano hueco o cavidad corporal.

Se habla de endoscopia rígida cuando en la aplicación de esta técnica se emplea un endoscopio no flexible. Pueden además introducirse instrumentos para realizar ciertas maniobras terapéuticas o biopsias. Entre las técnicas de endoscopia rígida más comunes encontramos la laparoscopia, la artroscopia y la colonoscopia.



Figura 2.3: Endoscopio rígido.

Cuando para ello se requiere un único endoscopio y orificio de entrada se denomina *endoscopia rígida monocanal*. Algunas técnicas de endoscopia rígida monocanal pueden ser la histeroscopia, la neuronavegación y ciertos tipos de artroscopia.

Este proyecto se centra concretamente en las técnicas de histeroscopia, si bien, desde el punto de vista tecnológico, es aplicable a cualquier tipo de endoscopia rígida. Así, es importante destacar que el software que se desarrolla en este proyecto es aplicable, con ligeras modificaciones, a la simulación virtual de cualquier tipo de técnica de endoscopia rígida.

Histeroscopia

La histeroscopia es un procedimiento clínico mediante el cual se visualiza el interior del útero por medio de un endoscopio, realizándose bien con fines diagnósticos, bien para tratar alguna patología o para realizar alguna intervención quirúrgica.

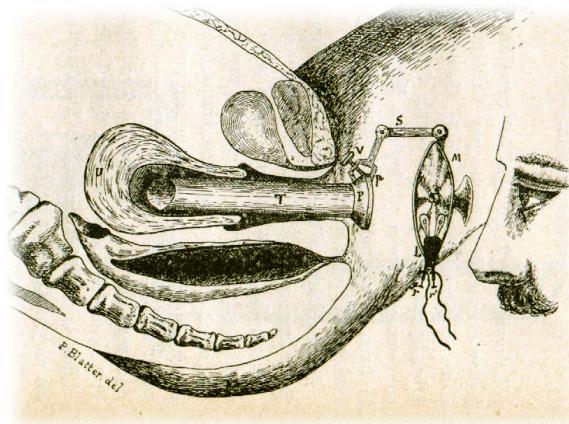


Figura 2.4: Boceto esquemático del procedimiento de histeroscopia diagnóstica.

Para ello, el endoscopio se guía a través del orificio cervical, accediéndose entonces a la cavidad uterina y posibilitando su exploración, diagnóstico y/o

intervención.

2.2. Simulación virtual

La simulación virtual se basa en el empleo de computadores y técnicas de modelado 3D, así como algoritmos varios, para la recreación de una experiencia concreta de manera que el usuario no encuentre diferencia práctica con la experiencia real correspondiente.

Los simuladores virtuales más conocidos probablemente sean los simuladores de vuelo. Estos simuladores son comúnmente utilizados como parte de la instrucción de los pilotos de vuelos comerciales y militares con el fin de entrenar su reacción ante diversos imprevistos. La complejidad de estos simuladores va desde la simple utilización de periféricos que se asemejan a los mandos de un avión estándar hasta la construcción de cabinas que son réplicas exactas de un modelo real.



Figura 2.5: Simulador de vuelo de la NASA, donde se recrea la cabina de un avión bimotor.

2.2.1. Simulación médica virtual

Se habla de simulación virtual médica o sanitaria cuando se aplica la tecnología de la simulación virtual a la simulación y entrenamiento de técnicas sanitarias. El concepto surgió en los años 80 con el auge de los videojuegos, pero no fue hasta la década del 2000 cuando la tecnología permitió su implementación.

Este concepto surge con el objetivo de facilitar la práctica de las técnicas sanitarias de mínima invasión. Estas técnicas, al acceder al paciente indirectamente,



Figura 2.6: Simulación virtual de una técnica laparoscópica.

mente a través de una interfaz tecnológica, simplifican el proceso de simulación virtual. A menudo la visualización se realiza por una pantalla y el instrumental puede sustituirse por periféricos especialmente diseñados para emular el comportamiento del original.



Figura 2.7: Simulación virtual de una ecografía.

Las ventajas son muchas, pues se permite el entrenamiento de las habilidades del especialista en un entorno controlado y que se comporta de manera muy similar al entorno real. Esto hace que se reduzcan los riesgos derivados del aprendizaje y también de ciertas intervenciones, ya que estos sistemas pueden utilizarse para el ensayo de una intervención concreta. Además se consigue acelerar el aprendizaje y se promueve el perfeccionamiento de las técnicas sanitarias, pues a priori no existen factores que limiten el tiempo de ensayo.

La simulación virtual sanitaria se aplica en Canarias a través del ITC y del Servicio Canario de Salud. En este contexto surgió el entorno ESQUI[1], con el que está estrechamente relacionado uno de los tutores de este proyecto, Miguel Ángel Rodríguez Florido, quien dirige actualmente las actividades del *Laboratorio de Simulación y Formación Basada en Tecnología del Hospital Universitario Insular Materno Infantil* de Las Palmas de Gran Canaria. Es

también en colaboración con este departamento que surge este proyecto.



Figura 2.8: Localización del *Laboratorio de Simulación y Formación Basada en Tecnología* en el Edificio Anexo del Hospital Insular de Gran Canaria.

2.3. Entorno ESQUI

ESQUI[1] es un entorno multiplataforma y de software libre diseñado para proveer las herramientas necesarias para la simulación virtual de técnicas sanitarias mínimamente invasivas. En él se da solución a las tres áreas principales de la simulación quirúrgica virtual:

- Diseño y generación de los modelos 3D de la escena quirúrgica.
- Software y algoritmia de la simulación.
- Administración de la información en la escena quirúrgica.

Para el área de diseño se emplea el software de modelado *Blender*[6], por ser un software de código libre, multiplataforma y de gran versatilidad. A fin de cubrir las otras dos áreas se implementó la librería vtkESQui y el formato SRML.

ESQUI surge con el objetivo de proporcionar una base para la creación de sistemas de simulación virtual de técnicas sanitarias de código libre y multiplataforma. Ello permitirá que el uso de estos sistemas se extienda y que los centros formativos dispongan de alternativas a los sistemas de simulación comerciales.

2.3.1. Licencia

ESQUI es distribuido bajo licencia *LGPL*[7]. Esta licencia permite la copia y distribución del material en cualquier medio o formato, además de la realización de alteraciones y la utilización del mismo para cualquier propósito —incluso comercial—, siempre que se haga mención de la autoría, la licencia

del material original y las alteraciones realizadas.

La licencia *LGPL* es de notable relevancia en cuanto a que casa con la filosofía del proyecto original y favorece la extensión y depuración del mismo. Así se consigue promover que el entorno alcance niveles de potencia y estabilidad superiores.

Por otro lado, la libertad de uso que provee esta licencia es garante para con el objetivo didáctico del proyecto, permitiendo que se adopte con mayor facilidad. Además, las empresas podrán hacer uso comercial del mismo, ya que la licencia *LGPL* no restringe el empleo del material con fines comerciales. No obstante, este uso comercial podría revertir en mejoras aportadas al entorno ESQUI, por parte de dichas entidades, producto de su experiencia de uso.

2.3.2. vtkESQui

La librería vtkESQui[8] implementa toda la algoritmia y gestión software necesaria para poner en marcha una simulación virtual. vtkESQui está escrita en C++ y dispone de Tcl y Python como lenguajes de scripting.

Una simulación virtual en vtkESQui se compone principalmente de un escenario —donde se incluyen todos los objetos virtuales que interactúan en la simulación y las características del entorno virtual— y de un proceso que coordina la interacción entre todos esos objetos.

Durante la simulación, vtkESQui da la posibilidad de calcular colisiones entre objetos, además de proveer distintos modelos para la simulación de deformaciones. También se da soporte a la interacción con distintos dispositivos periféricos.

2.3.3. SRML

SRML[1] (Surgical Reality Modeling Language) es un lenguaje de descripción de escenarios quirúrgicos basado en XML. Este formato permite detallar en un fichero todos los datos necesarios para una simulación y se concibió como un formato de descripción de simulaciones intercambiable y extensible. Es importante señalar que este formato **surgió de manera independiente al SRML de W3C[9]** y que la coincidencia de siglas es casual, pues entre ambos formatos no existe relación de concepto.

Mediante un fichero SRML se especifican las rutas de los modelos, los parámetros biomecánicos, las localizaciones espaciales de los objetos y otros parámetros de la escena virtual. Actualmente, un documento SRML de ESQUI se rige por una especificación DTD que se incluye en el directorio de fuentes del entorno. Esta especificación es, sin embargo, extensible de cara a contemplar las posibles mejoras del entorno ESQUI.

Capítulo 3

Análisis de los requisitos técnicos

En este capítulo se analizan los requisitos del proyecto desde el punto de vista técnico, pues ya se han aclarado los objetivos y se ha analizado el contexto del mismo.

3.1. Estudio del prototipo

En este proyecto se pretende implementar un prototipo funcional enfocado a la simulación de técnicas de endoscopia rígida monocanal. Dicho prototipo deberá mostrar las capacidades y el potencial de simulación del entorno ESQUI como resultado del trabajo en este proyecto. Así pues, se ha considerado que la aplicación debe cumplir las siguientes expectativas:

- **Debe ser multiplataforma.** Siendo ESQUI el entorno del que se parte para la implementación del prototipo y siendo ESQUI un entorno multiplataforma, no tiene sentido restringir la aplicación al uso en un determinado tipo de sistema operativo. Contando con esta versatilidad de base, lo óptimo es aprovecharla y transmitirla a la propia aplicación.
- **Ha de contar con una interfaz gráfica sencilla, intuitiva y práctica.** Esto es un requisito deseable en cualquier tipo de aplicación. La interfaz gráfica no debe distraer al usuario de su objetivo, sino facilitar la labor y hacer que la interacción sea lo más cómoda y natural posible.
- **El renderizado de VTK estará integrado en un elemento de la interfaz, a través del cual se visualizará la simulación.** La parte de código relativa a VTK deberá integrarse visualmente con el resto de la aplicación, de manera que el usuario tome la aplicación como un conjunto indivisible y no como una agrupación de elementos que interactúan entre ellos. Esto tiene relación directa con la sencillez, intuitividad y practicidad de la interfaz gráfica.

- A pesar de tratarse de un prototipo, tiene que contar con las características básicas de un simulador virtual de endoscopia rígida y para ello **debe poder simular los siguientes aspectos relativos a la misma:**
 - Visualización de órganos y herramientas.
 - Dinámica y comportamiento de un endoscopio rígido monocanal, incluyendo la visualización a través de una lente con cierto grado de inclinación.
 - Posibilidad de interacción con los órganos.
- **El prototipo tiene que poder cargar un escenario de simulación a partir de un documento en formato SRML.** Esto permitirá administrar las escenas con mayor facilidad. Además, posibilitará que el usuario cargue ejemplos predefinidos, así como futuras escenas que puedan ser añadidas.
- **El usuario podrá escoger la inclinación de la lente que desea utilizar durante la simulación.** La inclinación de la lente es una característica intrínseca de los endoscopios rígidos, por lo que simular tal aspecto se antoja indispensable. De cara al aprendizaje de la técnica, es importante que el usuario pueda familiarizarse con la visión que obtendrá mediante la utilización de distintos grados de inclinación en la lente.
- **Durante la simulación, las colisiones entre el endoscopio y los órganos han de ser representadas visualmente en pantalla.** Debido a que el feedback háptico no se plantea en este proyecto, es necesario que el usuario obtenga esta información a través de algún otro medio. En este caso, el medio visual servirá de alternativa para comunicar al usuario tales colisiones.
- Con el objetivo de replicar el comportamiento de un endoscopio rígido y así dotar a la aplicación de las capacidades docentes deseadas, **la interacción del usuario con la aplicación debe posibilitar las siguientes acciones:**
 - Movimiento del endoscopio virtual, tanto lateralmente, como verticalmente y en profundidad. El movimiento debe ser fiel a la dinámica real de un endoscopio rígido monocanal.
 - Selección y movimiento de una herramienta. La herramienta ha de poder moverse en profundidad y de poder accionarse si éste fuera el caso.
 - Rotación solidaria de la lente virtual y la herramienta si la hubiera.
- La interacción óptima tendrá lugar con un dispositivo periférico tipo joystick. Sin embargo, con fines de depuración y demostración, **la interacción debe ser posible también mediante teclado y ratón.**

- **En el caso de utilizarse un dispositivo periférico tipo joystick, la aplicación debe facilitar la interacción a través de este medio y anular la interfaz de ratón/teclado.** Así se conseguirá que el usuario se centre en la técnica de endoscopia rígida y no se distraiga con elementos ajenos a la misma (ratón y teclado).
- **La instalación de la aplicación ha de ser trivial.** Todas las dependencias han de ser mitigadas en la medida de lo posible. El modelo de usuario final de este prototipo no tiene conocimientos especiales relativos al ámbito de los computadores, de manera que facilitando estos aspectos se conseguirá una mejora en la experiencia y que el usuario no se encuentre con trabas durante la instalación y ejecución de la aplicación.

En base a los criterios anteriormente citados, se procede ahora a analizar los requisitos técnicos de cara a la implementación de este prototipo.

3.2. Dispositivo periférico

Una característica esencial en las simulaciones virtuales es la de la interfaz hombre-máquina. El concepto en sí requiere que la interacción sea lo más cercana posible a la realidad y es aquí donde juegan un papel importante los dispositivos periféricos diseñados para emular el entorno real de interacción.

Un dispositivo háptico es aquel que permite una interacción con la máquina basada en el sentido del tacto y el movimiento de las manos. La manipulación por parte del usuario de un joystick o similar proporciona los datos de entrada para la interacción en la simulación. Por otro lado, el dispositivo háptico también provee mecanismos de feedback táctil que dan al usuario información acerca de la escena: colisiones, dureza de un elemento, resistencias, etc.

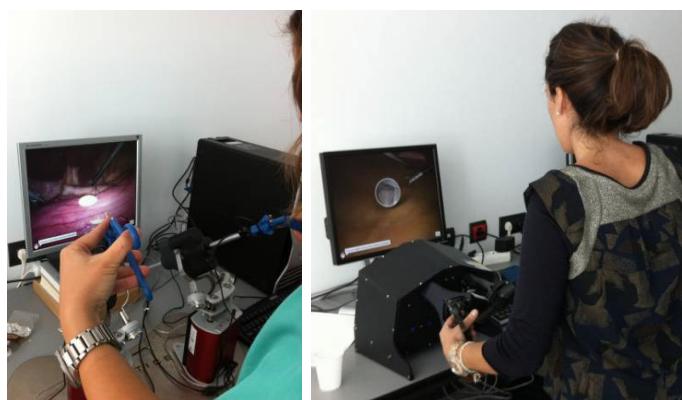


Figura 3.1: De izquierda a derecha, sistemas hapticos *Xitact IHP* e *Immersion LSW*.

En el entorno ESQUI se implementan interfaces para tres tipos de periféricos con soporte háptico: *Xitact IHP*, *Xitact VSP*[10] e *Immersion Laparoscopic Surgery Workstation (LSW)*[11][12]. No obstante, en el *Laboratorio de Simulación y Formación Basada en Tecnología del Hospital Universitario Insular Materno Infantil* se disponía también de acceso al joystick *Simball 4D* del fabricante *G-coder*[3].

Esta ocasión motivó que se incluyera en el proyecto la implementación de una nueva interfaz en la que se diera soporte a este dispositivo. El *Simball 4D*, a pesar de no proporcionar feedback táctil, sí cumple con las necesidades de interacción básicas, permitiendo que el usuario interactúe en tiempo real y con una mecánica similar a la que proporcionaría la utilización de un endoscopio rígido.



Figura 3.2: Utilización del sistema *Simball 4D*.

El *Simball 4D* utiliza un sistema de codificación óptica para detectar la orientación del trocar. Esto hace que sea innecesario realizar cualquier tipo de calibración del dispositivo.

Por otro lado, fue posible tener acceso a la API del dispositivo, si bien estaba implementada únicamente para plataformas Microsoft Windows. Esto limita el aspecto multiplataforma del prototipo, pero tecnológicamente puede adaptarse al uso de cualquier dispositivo periférico soportado por ESQUI.

3.3. Herramientas de desarrollo software

Puesto que el entorno de desarrollo será multiplataforma, las alteraciones que sufra tanto la plataforma vtkESQui como la aplicación deberán funcionar correctamente en sistemas Microsoft Windows y UNIX. En base a estos criterios de compatibilidad, este proyecto se sirve de las siguientes herramientas de desarrollo.

3.3.1. CMake

CMake[13] es un sistema multiplataforma cuya función es la de facilitar la compilación, prueba y empaquetado de software. Se emplea para controlar el proceso de compilación utilizando ficheros de configuración independientes del compilador y la plataforma. Surgió de la mano de Kitware como solución a la necesidad de un entorno de compilación potente y multiplataforma para proyectos de código libre como VTK.

Tanto VTK como ITK son proyectos de software libre de gran envergadura desarrollados por Kitware y su proceso de compilación se controla mediante CMake. Es lógico entonces que vtkESQui emplee el mismo sistema ya que es una extensión de VTK.

3.3.2. Doxygen

Doxygen[14] es un generador de documentación que se ha adaptado a los lenguajes de programación más extendidos. Es la herramienta de este tipo más usada en fuentes C++. Mediante el sistema de Doxygen, la documentación va incluida en los comentarios del código con una sintaxis específica. Así es relativamente sencillo mantenerla actualizada. Luego se puede generar la documentación en el formato deseado, que pueden ser documentos navegables (HTML) o no (L^AT_EX, PDF, RTF...).

Si bien la documentación de VTK se genera mediante un sistema propio distinto a Doxygen, sí que sigue una metodología similar, estando la documentación embebida en los comentarios del código y generándose la misma en tiempo de compilación. En vtkESQui se prefirió utilizar Doxygen porque era una herramienta de software libre disponible que cumplía con las necesidades de documentación del proyecto y, por lo tanto, sería ilógico y contraproducente no utilizarla para dedicar esfuerzos en este aspecto.

3.3.3. Python

Python[5] es un lenguaje de programación interpretado de tipado dinámico, multiplataforma y multiparadigma. Su función principal es la de lenguaje de scripting, pero también se utiliza para otros fines apoyándose en librerías

de terceros.

Python soporta programación orientada a objetos, imperativa y funcional. La legibilidad y la compacidad de código son sus características más destacables y la gestión de la memoria se realiza de forma automática. Es ampliamente conocido por sus librerías estándar y por contar con una notable variedad de librerías externas.

La licencia de código abierto que posee —llamada *Python Software Foundation License*[15]— es compatible con la licencia pública general GNU, lo que ha propiciado su aceptación por parte de la comunidad de código libre y su uso en VTK como lenguaje de scripting.

Mediante el uso de herramientas de terceros como *Py2exe*, *Py2app* o *Pyinstaller* es posible empaquetar código Python en un ejecutable autocontenido, de manera que el usuario final no tenga que realizar ninguna instalación ni preocuparse de las dependencias que pueda haber.

La versatilidad de este lenguaje de programación, su potencia, la amplia funcionalidad proporcionada por la gran cantidad de librerías disponibles y el hecho de contar con previa experiencia en su uso fueron los factores que decantaron hacia su utilización como lenguaje principal de la aplicación. Las herramientas empaquetadoras serán de gran utilidad para eliminar dependencias y disponer de cara al usuario de un ejecutable independiente.

3.3.4. GIT

GIT[16] es una herramienta de control de versiones especialmente diseñada para proyectos de software. Permite la creación y unificación de ramas de desarrollo, posibilitando deshacer los cambios en el caso de que fuera necesario.

Al contrario que otras herramientas del mismo tipo, GIT da la opción de trabajar en local, sin necesidad de conexión con el servidor de versiones, hasta el momento en que el usuario solicite la sincronización. Además, se caracteriza por su rapidez y eficacia en las operaciones cotidianas como el registro de cambios y la resolución de conflictos.

El proyecto de la plataforma vtkESQui se encuentra alojado en un servidor de versiones GIT[8] y para este proyecto en particular se creó un repositorio adicional[17].

3.3.5. VTK

VTK (*Visualization ToolKit* o conjunto de herramientas de visualización) es un sistema de software libre creado por *Kitware Inc.*[2] y que está orientado a la realización de gráficos 3D por computadora, procesamiento de imagen y

visualización.

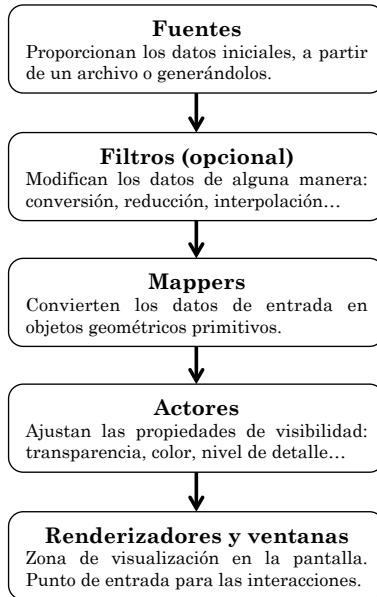


Figura 3.3: Pipeline de visualización de VTK.

El código de VTK está implementado en C/C++ y cuenta con capas de lenguaje interpretado para Tcl/Tk, Java y Python. Para el desarrollo de este proyecto se utiliza la versión 5.10 del kit debido a ciertas incompatibilidades que se encontraron con la capa interpretada en Python.

3.3.6. vtkESQui

Esta librería implementa toda la algoritmia y gestión software necesaria para poner en marcha una simulación virtual. La librería vtkESQui^[1] ha sido construida como una extensión de la librería VTK.

El código funcional de VTK está escrito en C++, por lo que vtkESQui está escrita en C++ y dispone de Tcl y Python como lenguajes de scripting —Java es el único que sí tiene soporte en VTK pero todavía no se ha implementado en vtkESQui.

Una simulación virtual en vtkESQui se compone principalmente de un escenario, donde se incluyen todos los objetos virtuales que interactúan en la simulación y las características del entorno virtual, y de una clase *vtkSimulation* que coordina la interacción entre todos esos objetos. Para comprender en mayor detalle el funcionamiento de la librería, desgranaremos cada uno de sus módulos principales. Además, la estructura de directorios del código de la li-

brería es muy descriptiva, así que la tomaremos como punto de partida para explicar el funcionamiento de cada uno de los módulos.

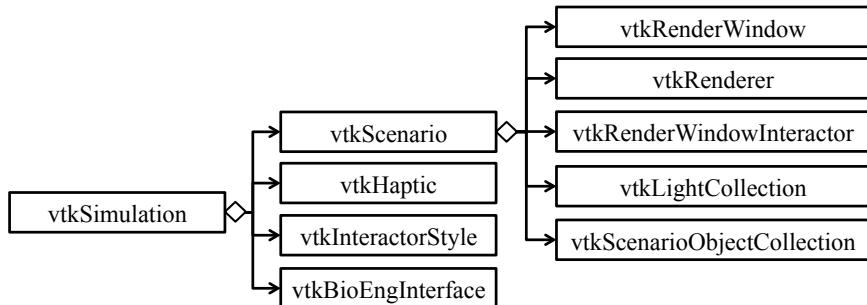


Figura 3.4: Diagrama que muestra las clases principales de `vtkESQui` y sus agregaciones.

Módulo de simulación

En este módulo se declara la clase `vtkSimulation`, que se encarga de contener todos los elementos de la simulación y coordinar su funcionamiento. A través de ella se configuran los parámetros principales como la frecuencia de renderizado, la activación de colisiones y el estilo de interacción.

Módulo de escenario

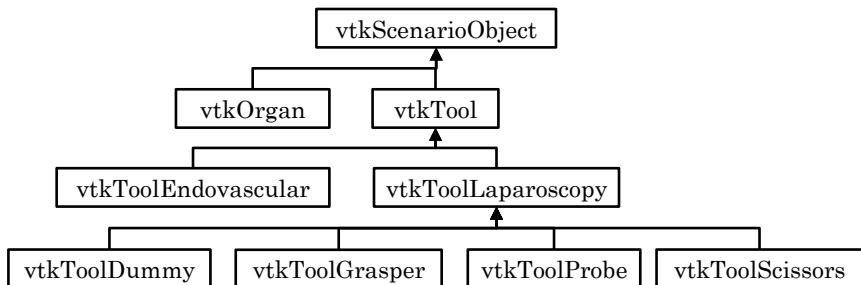


Figura 3.5: Gráfico de herencia de las clases en un escenario de `vtkESQui`.

Este módulo contiene las clases que representan los objetos de una simulación virtual y sus componentes. Todos los objetos de una simulación virtual heredan de la clase `vtkScenarioObject` y están compuestos por elementos (`vtkScenarioElement`). A su vez, cada elemento está formado por hasta tres modelos:

- *Modelo de visualización.* Es el modelo que no puede faltar en un elemento. En él se determina la geometría de la malla que será visible y otras

características visuales como el color, la textura y su opacidad.

- *Modelo de colisión.* Este modelo opcional suele tener una geometría de menor resolución y es el que se emplea para el cálculo de colisiones entre objetos.
- *Modelo de deformación.* Un elemento con modelo de colisión puede tener además un modelo de deformación, que incluye un modelo biomecánico para el cálculo de las deformaciones en base a las colisiones detectadas. Esta deformación puede configurarse en base a unos parámetros que determinan el comportamiento del tejido simulado.

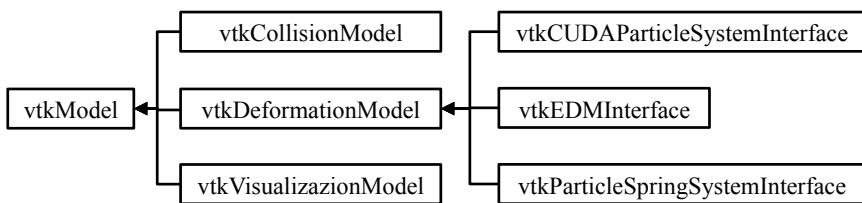


Figura 3.6: Gráfico de herencia de los distintos modelos que contempla vtkESQui y algunos modelos de deformación.

La geometría de un modelo puede especificarla el programador a partir de un objeto de fuente geométrica (*vtkPolyDataSource*) o, como es común, importarla desde un archivo *vtp*[18] proporcionando al modelo el nombre del fichero.

De entre todos los objetos de un escenario, se hace diferencia entre objetos que representan un elemento orgánico (*vtkOrgan*) y los que representan una herramienta (*vtkTool*). Es importante marcar esta diferencia, porque las herramientas suelen tener métodos específicos en base a su funcionalidad —como la apertura de una pinza en la herramienta *vtkToolGrasper*— y sus métodos de transformación pueden tener efectos distintos dependiendo del tipo de la herramienta.

Las herramientas virtuales se modelan en vtkESQui mediante la clase *vtkTool*. *vtkTool* hereda de *vtkScenarioObject* y es una clase abstracta que implementa la base para la construcción de cualquier herramienta virtual bajo vtkESQui. Así, esta clase conforma una interfaz común para trabajar con herramientas virtuales de vtkESQui, dando información del tipo de herramienta que representa.

Cada uno de los tipos de herramientas virtuales se crean heredando de esta clase. Esta discriminación se hace necesaria porque cada tipo de herramienta tiene su propia mecánica y características comunes. Por poner un ejemplo, una herramienta de tipo endovascular es flexible, mientras que una de tipo

laparoscópico es rígida y sus características mecánicas son, por ende, diferentes. Cada tipo de herramienta virtual puede implementar también métodos concretos que satisfagan las peculiaridades del mismo.

Las clases que representen herramientas concretas —como lo es *vtkToolGrasper*— heredan del tipo de herramienta al que pertenecen e implementan los métodos necesarios para definir las peculiaridades que requiera la herramienta representada.

Módulo de detección de colisiones

Éste es el módulo donde se engloban las clases orientadas a la detección y manejo de colisiones. *vtkESQui* proporciona para ello una clase interfaz (*vtkBioEngInterface*) que se encarga de todo el proceso, recogiendo los modelos de colisión que van a intervenir en el cálculo de colisiones y las características y atributos del mismo. Para el cálculo de colisiones emplea un filtro de colisiones (*vtkCollisionDetectionFilter*) y hereda de la clase *vtkCollisionDetection*, cuya principal función es mantener un registro de dichas colisiones.

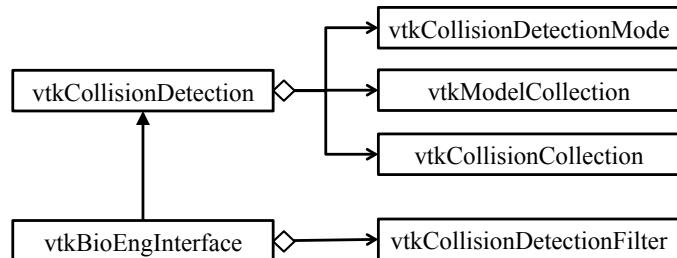


Figura 3.7: Gráfico de herencia de *vtkBioEngInterface*.

Módulo de biomecánica

El módulo de biomecánica congrega los distintos métodos y clases que se emplean para simular la mecánica de los tejidos. Incluye código para el cálculo de las deformaciones a partir de un modelo de masa-muelle.

Módulo de interacción

Aquí vienen definidos los estilos de interacción propios de *vtkESQui*. En VTK, un estilo de interacción (*vtkInteractorStyle*) es una clase que es capaz de determinar qué acciones se llevan a cabo ante cada uno de los eventos lanzados a través del interactor. Dichos estilos están enfocados a posibilitar la interacción con la simulación aún cuando no haya ningún dispositivo periférico conectado. Normalmente, esto ocurre cuando se desea hacer pruebas mientras

se desarrolla la simulación.

Módulo de soporte periférico

En vtkESQui, los dispositivos periféricos interactúan con la escena a través de una clase orientada a tal efecto: *vtkHaptic*. Puesto que existen distintos tipos de dispositivos con sus respectivos controladores e interfaces, es esta clase abstracta la que unifica la funcionalidad básica del dispositivo periférico para que la simulación pueda hacer uso de él de manera homogénea.

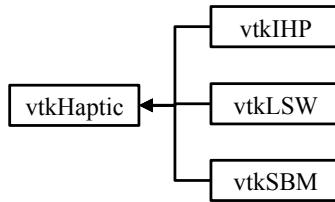


Figura 3.8: Gráfico de herencia de la clase vtkHaptic

3.3.7. wxWidgets

wxWidgets[4] es una librería de código libre escrita en C++ que permite crear interfaces gráficas multiplataforma. Esta librería tiene la peculiaridad de que emplea la API nativa del sistema en el que se ejecuta, dando como resultado una interfaz de aspecto y comportamiento nativo.

También se pueden encontrar adaptaciones para Python, Perl, Ruby y otros lenguajes interpretados. wxPython[19] es la adaptación de la librería wxWidgets para Python. Está implementada como un módulo de extensión en código nativo que wrapea la librería wxWidgets.

Debido a la esencia multiplataforma de la librería wxWidgets y a sus resultados de aspecto y comportamiento nativo se decidió tomar wxPython como la librería para construir la interfaz gráfica de la aplicación prototipo del proyecto.

Capítulo 4

Planificación del proyecto

El desarrollo de este proyecto se fundamenta en una serie de aspectos tecnológicos que es necesario controlar previo a la implementación de cualquier característica del prototipo.

Por un lado, las herramientas de desarrollo que se utilizan disponen de cierta complejidad que conviene acotar. Así, la planificación contempla una primera fase de aprendizaje en la que se persigue la familiarización con las herramientas de desarrollo pertinentes.

Por otro lado, el proceso de implementación del prototipo puede y debe subdividirse en tareas más concretas que marquen un desarrollo progresivo de la tarea general y, sobre todo, que discriminen los distintos procesos de desarrollo.

4.1. Formación en la tecnología de desarrollo

En este primer periodo, de carácter formativo, se persigue la adquisición de una serie de conocimientos básicos sobre la tecnología de desarrollo que se emplea en el proyecto. El dominio de la tecnología necesaria es vital de cara a un desarrollo correcto de la fase de implementación.

Las distintas tecnologías de desarrollo se apoyan unas en otras para conseguir el resultado fijado, de manera que la tecnología más específica se sirve de la más genérica. Desde el punto de vista formativo se sigue este mismo enfoque, abordando en primer lugar la tecnología más genérica para luego ir progresando hacia la especificidad requerida.

Tomando este criterio como guía, la primera tecnología en abordarse es la librería de visualización VTK[2], que constituye la base del soporte gráfico de la librería vtkESQui[1] y, por ende, de la aplicación prototipo. Además, vtkESQui es una extensión de VTK, por lo que la estructura y funcionamiento

de ambas librerías se rige por los mismos patrones. El objetivo es instalar la librería VTK, comprender su flujo de ejecución y aprender su uso a nivel de usuario.

Una vez dominado este punto, el siguiente paso es familiarizarse con la capa de lenguaje interpretado Python de la librería. Esta capa es la que se utiliza en la aplicación y es por tanto con la que hay que familiarizarse. Para ello, se realizan ejemplos sencillos en los que se crean y gestionan objetos primitivos.

La herramienta de configuración de proyectos CMake[13] es utilizada para la configuración de los proyectos de las librerías VTK y vtkESQui. Es por esto que se consideró conveniente establecer el aprendizaje de los conceptos básicos de uso de esta herramienta como siguiente punto en la planificación. Se realizaron ejemplos sencillos con el objetivo de asentar los conocimientos adquiridos.

Finalmente, la instalación, análisis y aprendizaje de la estructura y funcionamiento de la plataforma de desarrollo en simulación vtkESQui[1] permite obtener una imagen genérica del punto de partida para el desarrollo de la aplicación prototipo de simulador virtual de técnicas de endoscopia rígida monocanal. Con este objetivo se realizaron ejemplos sencillos, en los que también se tuvo en cuenta el acceso a los sistemas periféricos.

4.2. Desarrollo tecnológico

Una vez adquiridos los conocimientos imprescindibles acerca de la tecnología de desarrollo necesaria, es posible comenzar la fase de desarrollo tecnológico. Esta fase se centra en la creación del software que satisfará los objetivos del proyecto y consta de los siguientes puntos:

1. Identificación de las herramientas disponibles en el entorno de simulación básico para el desarrollo de un simulador para técnicas endoscópicas. Detección de posibles dificultades y nuevos desarrollos software necesarios para abordar el proyecto.
2. Generación de mallados 3D sencillos que sirvan de entrada y escenario para la realización de pruebas de concepto.
3. Definición de las especificaciones necesarias para la entrada de datos e interfaz de usuario. Realización de ejemplos sencillos que permitan evaluar las opciones. Crear un proyecto GIT de versiones y de documentación del software para comenzar su utilización a lo largo del proyecto.
4. Implementación del prototipo de simulador. Carga de escenarios sencillos, interacción con el sistema periférico y simulación de la navegación 3D básica de un endoscopio rígido monocanal.

5. Identificación del instrumental específico de alguna técnica concreta de endoscopia rígida monocanal para introducir instrumental adicional a los ejemplos proporcionados con el prototipo.

4.3. Validación y documentación

En última instancia, y una vez implementada la aplicación prototipo de simulador virtual de técnicas de endoscopia rígida monocanal, se intenta hacer especial énfasis en su faceta docente.

La validación del prototipo por parte de personal especializado en la técnica escogida es vital para la aplicación docente del prototipo. Mediante este proceso, se corrigen errores que el desarrollador haya podido pasar por alto debido a su perfil técnico y se aplican mejoras y refinamientos para acercar la simulación aún más a la experiencia real.

No obstante, esta etapa no pudo llevarse a cabo por razones de carácter logístico y temporal, ya que no surgió la posibilidad de establecer coordinación con un especialista. Además, se observó que dicho proceso excedía las limitaciones temporales del proyecto.

Para concluir, se destina un plazo de tiempo exclusivo para la generación y revisión de la documentación del prototipo, así como del texto de este proyecto y para la preparación de su defensa.

Capítulo 5

Implementación del proyecto

Para la implementación correcta del prototipo, en primer lugar es necesario trabajar primero sobre vtkESQui, que es el núcleo del mismo pues soporta el núcleo de la simulación virtual. Luego, el trabajo se centra sobre la interfaz de usuario y ciertos mecanismos de control a nivel de scripting.

Durante las fases de desarrollo tecnológico del proyecto se detectaron una serie de deficiencias y carencias en la librería de vtkESQui. En este apartado se describen las mismas y se explica el proceso de depuración e implementación, para posteriormente comentar el proceso de implementación de la interfaz gráfica y el control del flujo de la aplicación.

5.1. Metodología de desarrollo

Parte de los objetivos de este proyecto se centran en depurar y ampliar la librería vtkESQui. Tanto los objetivos en sí como el hecho de que vtkESQui sea una ampliación de VTK, de código libre y con una filosofía de desarrollo muy concreta, obligan a emplear un ciclo de vida personalizado y distinto de los métodos clásicos.

En primer lugar, y debido a que los procesos de depuración son poco predecibles en cuanto a tiempo y complejidad, el método de desarrollo ha de ser ágil para poder adaptarse a los posibles imprevistos. Por otro lado, vtkESQui cuenta con una estructura del software que está suficientemente definida y permite enfocar la metodología a el desarrollo de los objetivos concretos. Además, como vtkESQui se desarrolla en torno al paradigma de la programación orientada a objetos, muchos de los requisitos del software se ven reflejados en objetos —en el sentido de la programación orientada a objetos— cuya interfaz para con el resto de la librería ya está definida, lo cual simplifica la fase de diseño y permite centrarse en el desarrollo individual y progresivo de estos objetivos técnicos.

Así pues, y en base a lo comentado anteriormente, se decidió seguir un ciclo de vida del software híbrido, tomando las características más deseadas

de los ciclos estandarizados[20]:

- **Programación extrema (XP).**

VTK está soportado por una gran comunidad de desarrolladores que, en cierta forma, promueve los valores de esta metodología: comunicación, sencillez, retroalimentación, valentía y respeto. Estos son valores muy presentes en la comunidad de desarrollo del software libre y permiten que estas comunidades de cientos o miles de programadores puedan trabajar en conjunto.

En este ámbito se hace énfasis en el primer aspecto: el de la comunicación. Así, se establece que el código ha de ser legible y estar bien comentado. La documentación se integra en el código para proporcionar un extra de flexibilidad.

- **Desarrollo basado en componentes.**

vtkESQui, como ampliación de VTK que es, se apoya sobre el código de VTK. Así, todos los objetos de vtkESQui heredan de clases de VTK y, en casi todos los casos, reutilizan gran parte de los métodos heredados. Esta posibilidad de reutilizar grandes cantidades de código potencia y facilita el desarrollo de nuevas funciones. No se encuentra, sin embargo, la mayor desventaja de esta metodología, ya que al tratarse de software libre cualquier parte del mismo es utilizable y ampliable.

- **Construcción de prototipos y desarrollo rápido de aplicaciones.**

Por último, la relativa independencia a nivel de software de los requisitos técnicos permite un desarrollo aislado y modular de los mismos, produciéndose prototipos para la prueba de cada módulo y cambio. Todo esto con vistas a una posterior integración y la construcción de prototipos destinados a probar y mejorar la aplicación a nivel general.

5.2. Código en C/C++ y clases de vtkESQui

La librería vtkESQui se encontraba en una versión con una estabilidad y funcionalidad limitada. Las pruebas que se habían realizado sobre ella habían sido fundamentalmente modulares y de integración básica, principalmente orientadas a la simulación virtual de técnicas de laparoscopia.

Esto motivó no sólo el hecho de que se realizaran labores de depuración de la librería, sino también la extensión y mejora de ciertos módulos y aspectos de la misma. Se partirá de la estructura de módulos por directorios de vtkESQui para explicar las modificaciones realizadas.

5.2.1. Módulo de simulación

La clase *vtkSimulation* es la encargada de ofrecer los mecanismos para la puesta en marcha de una simulación quirúrgica y la correcta coordinación de todos sus elementos. Se encontraron en ella una serie de defectos y aspectos que podían ser mejorados.

Optimizaciones en la inicialización

Se observó que en la versión de partida de *vtkESQui*, en el método de inicialización, no se comprobaba si el objeto había sido previamente inicializado. Esto podría resultar en fallos poco previsibles y un consumo de recursos del sistema innecesario. En la versión actual se hace uso del flag de inicialización a tal efecto.

Además, también se corrigió la ausencia de comprobación de inicialización previa a la puesta en marcha del bucle de simulación, lo que ocasionaba fallos de página si se olvidaba tal acción.

Proceso de simulación

En primera instancia, la simulación se llevaba a cabo con la ejecución simultánea de tres bucles que realizaban, cada uno, una acción distinta:

- *Interact*: Se encarga de la actualización del instrumental virtual a partir de la información de los dispositivos periféricos.
- *Step*: Ordena el cálculo de colisiones y deformaciones y actualiza los objetos de la escena en consecuencia. Éste es el método que realiza el grueso de operaciones de la simulación.
- *Render*: Genera y actualiza la imagen de la escena.

Cada uno de estos bucles era en realidad una función que se activaba con la señal de un temporizador periódico. Cada bucle tenía su propio temporizador con su propia tasa de activación. Observé que esta práctica era poco eficiente y poco fiable.

Por un lado, los temporizadores son recursos limitados en un sistema operativo. Teniendo esto en cuenta, disponer de tres temporizadores no es lo ideal y sería mejor utilizar uno solo. Por otro lado, el hecho de que cada acción se pueda realizar a intervalos diferentes conlleva los siguientes problemas:

- **Cálculos redundantes**. Las tres acciones principales del mecanismo de simulación son interdependientes y la diferencia de intervalos da pie a que una de estas acciones se realice varias veces pero que sólo sea de utilidad en su primera ejecución. Por ejemplo —y esto se veía frecuentemente en los ejemplos antiguos— si la imagen que ve el usuario se genera cada

20ms, no tiene sentido calcular las colisiones a cada milisegundo. El efecto de esas colisiones sólo se apreciará cuando se genere la imagen y, por lo tanto, de las 20 iteraciones se estarían realizando 19 en vano.

- **Asincronía.** Debido a la interdependencia de las acciones de simulación puede darse el caso de una descoordinación entre ellas. La dependencia es la siguiente: la acción *Step* actúa en función de las alteraciones generadas por la interacción del usuario a través del método *Interact* y dichos resultados no se apreciarán hasta la actualización de la imagen que el usuario visualiza a través del método *Render*. Si, por ejemplo, los métodos *Render* e *Interact* se ejecutaran a mayor frecuencia que el método *Step*, el usuario podría visualizar estados del escenario incoherentes, tal como un instrumental que atraviesa un órgano sin deformación alguna.

Debido a esto se decidió que lo mejor era utilizar un sólo bucle que coordinara explícitamente las tres acciones principales y que, por ende, evitara tanto cálculos redundantes como inconsistencias de la escena. Tras la aplicación de esta medida, se utiliza un sólo temporizador que activa una función a fin de realizar las siguientes acciones en el siguiente orden: *Interact*, *Step* y por último *Render*.

Con esto se consigue que, al eliminarse los cálculos redundantes, la simulación sea más eficiente y puedan emplearse tasas de refresco mayores. Así el usuario experimenta mayor fluidez y se evitan las incoherencias ya mencionadas.

Estilo de interacción con la simulación

En la versión de partida de vtkESQui, los métodos *Set* y *Get* relativos al estilo de interacción limitaban éste al estilo por defecto (*vtkDefaultInteractorStyle*) o a herencias del mismo, cuya interacción corresponde a las técnicas de laparoscopia donde la cámara se introduce por un canal separado. Esto tiene origen en las primeras implementaciones de vtkESQui, donde se tomaba como referencia de trabajo dicha técnica.

Sin embargo, el objetivo es que la librería pueda ser utilizada para simular distintas técnicas, por lo que conviene aplicar el menor número de restricciones que sea posible. Así, ahora puede asignarse a la simulación cualquier tipo de estilo de interacción que herede del tipo base de VTK *vtkInteractorStyle*.

La parte negativa de este cambio es que si se emplea algún estilo de interacción que requiera de inicialización —como es el caso del estilo *vtkDefaultInteractorStyle* de vtkESQui (ver sección 5.2.5)— sería responsabilidad del programador y/o de la instancia del estilo de interacción llevar a cabo la inicialización en el momento que le corresponde. Sin embargo, se implementaron una serie de

controles durante la inicialización que tienen en cuenta estos aspectos, de manera que se libera al programador de esta responsabilidad y se evitan errores humanos en relación a este punto.

5.2.2. Módulo de escenario

En las primeras pruebas se observó que si un elemento no tenía asociado un modelo de deformación éste no se actualizaba automáticamente con la actualización del escenario. Esto obligaba a ordenar una actualización explícita de los objetos que hubieran sido alterados, lo que dificultaba y entorpecía la manipulación de la escena.

Por causas que ignoro, en la versión previa a este proyecto se empleaba este criterio en el método *Update* de *vtkScenario*. La solución fue trivial y consistió en eliminar la condición que evitaba actualizar los elementos que no tuvieran asignado un modelo de deformación, actualizando ahora todos los elementos del escenario.

Además, en una revisión rutinaria del código se descubrió que, durante la inicialización del escenario, el interactor se llamaba a inicializar multitud de veces, pues la orden se había escrito dentro del bucle de inicialización de los objetos de la simulación. Dicha orden se sitúa ahora fuera del bucle, ejecutándose una sola vez durante la inicialización del objeto *vtkScenario* —como es correcto que sea.

Desarrollo de herramienta virtual para endoscopia rígida monocanal

Los tipos de herramientas de los que disponía *vtkESQui* eran, en principio, *vtkToolEndovascular* para simular herramientas de tipo endovascular y *vtkToolLaparoscopy* para herramientas de naturaleza laparoscópica.

En principio se pensó que el endoscopio rígido monocanal se podría implementar con base en el tipo laparoscópico, pero más adelante se vio que era preferible crear un nuevo tipo para dar soporte a las peculiaridades mecánicas de la endoscopia rígida monocanal.

El hecho de que la herramienta se introduzca por un canal paralelo al de visionado condiciona el movimiento de la misma. A diferencia de las herramientas de laparoscopia, no giran con respecto a un origen situado en la misma herramienta, sino que lo hacen con respecto a un punto no concéntrico (ver figura 5.5). La utilización de esta nueva clase simplifica, principalmente y entre otras cosas, el código del estilo de interacción.

5.2.3. Módulo de detección de colisiones

En el proceso de integración y depuración de vtkESQui se encontró un defecto involucrado en el proceso de detección de colisiones y aplicación de deformaciones que causaba un fallo de página y abortaba la ejecución.

La fuente del problema se localizó primeramente en *vtkScenarioElement*, donde se accedía a elementos de un vector tipo *vtkIdList* sin hacer chequeo de rango. Sin embargo, esto no abortaba la ejecución, sino que sucedía más tarde al devolverse un número inválido y cuando otra función intentaba hacer uso del mismo.

El error se originaba en la inicialización de SynchronizationMap en *vtkBioEngInterface*. Se utilizaba el número de puntos de entrada del modelo de colisión, cuando luego se trabajaba con los puntos de salida del filtro de detección de colisiones. El filtro de detección de colisiones utiliza árboles para dividir el espacio de colisión del objeto, por lo que el número de puntos de su salida suele ser variable.

Contexto del problema

El proceso de detección de colisiones de vtkESQui es controlado por la clase *vtkBioEngInterface*. Esta clase se encarga de recordar los modelos que se verán involucrados en el cálculo de colisiones y de registrar en cada uno de ellos las colisiones que sufren en un ciclo de simulación.

Tal como se explicó en el apartado 3.3.6, un elemento del escenario puede tener hasta tres modelos. En el caso de que estén los tres presentes, tanto el modelo de visualización como el modelo de colisión se sincronizan con el modelo de deformación, lo que hace que el usuario pueda apreciar las deformaciones de manera visual pero que también se vean reflejadas en el modelo de colisión de cara a mejorar el comportamiento de futuras colisiones. Esto no era así en principio, pero se modificó para evitar problemas de sincronización.

La clase *vtkBioEngInterface* forma parte del módulo de detección de colisiones de vtkESQui. Para poder calcular las colisiones, el filtro de colisiones que maneja la clase asigna cada modelo de colisión a la entrada de un árbol de división espacial implementado en la clase *vtkOBBTree*. Estos árboles realizan una división espacial de manera recursiva y el número de divisiones viene dado por la morfología del objeto, por lo que el número de puntos no está acotado. Una vez hecho esto, calcula las colisiones entre ambos árboles, devolviendo los identificadores de estos puntos y no de los del modelo de colisión.

Solución aplicada

Como el número de puntos de un *vtkOBSTree* no está acotado, no queda otra alternativa que realizar un mapeo de puntos dinámico. A pesar de ello, si tenemos en cuenta que el número de puntos sólo cambia tras aplicar una deformación, se puede reducir el número de cálculos manteniendo un buffer con los valores previamente calculados y ampliándolo sólo tras cada deformación.

En la clase *vtkBioEngInterface* se mantienen dos buffers de mapeo de puntos por cada modelo de colisión. El primer buffer es constante tras la inicialización de la clase y asigna a cada punto del modelo de colisión el punto más cercano en el modelo de deformación. El segundo buffer se utiliza para asignar a cada punto del *vtkOBSTree* un punto en el modelo de colisión.

Este último buffer es dinámico y se emplea de la manera siguiente:

1. Inicialización. Para evitar costosos redimensionamientos del vector, se estima un tamaño máximo para la reserva de memoria. Los primeros valores se inicializan a partir de una colisión de prueba y el resto se inicializa con valores negativos.
2. Sincronización. Partiendo de un identificador de punto del *vtkOBSTree* se consulta su correspondencia en el buffer dinámico. En caso de corresponderse a un valor positivo sólo queda trasladar esa correspondencia al modelo de deformación mediante el buffer constante. Por contra, si el valor es negativo significa que el modelo ha sido alterado y hay nuevos puntos en el *vtkOBSTree*. Se requiere, por tanto, una ampliación del buffer.
3. Ampliación. En caso de que el *vtkOBSTree* altere su número de puntos es necesario sincronizar los puntos adicionales con el modelo de colisión. Gracias a la implementación recursiva del árbol es posible obviar la actualización de los puntos anteriores y sólamente dar valor a los puntos añadidos.

Además se tuvo esto en cuenta para la actualización de la clase *vtkCollision*, contemplándose ahora un campo que almacena el identificador de punto para la malla de deformación (*vtkDefPointId*) que complementa al que guarda el valor del identificador de punto para la malla de colisión (*vtkPointId*). Se decidió hacerlo así, en lugar de simplemente sustituir el valor del campo antiguo, porque este campo se ve involucrado en múltiples partes del código de *vtkESQui* y de esta manera se evitan efectos colaterales.

Direccionalidad

En el cálculo de colisiones de *vtkESQui* no se tiene en cuenta la direccionalidad de la misma. Es decir, que producirá el mismo efecto una colisión

que recibe un objeto desde el exterior que una que recibe desde su interior. Esto toma mayor relevancia cuando se tiene en cuenta que en una simulación interactúan órganos (que reciben colisiones desde el exterior) y cavidades (que las reciben desde el interior).

Sin embargo, de cara a producir una deformación en los objetos colisionantes, es necesario determinar la direccionalidad de alguna forma. En este caso se toman las normales de los polígonos para dar valor a la dirección de la deformación. Es por ello necesario conceder cierta importancia al sentido de las normales.

En el transcurso de este proyecto se detectó que el cálculo producía deformaciones en el sentido contrario a lo que sería idóneo, pues las normales de los objetos (que apuntan hacia el exterior del objeto) sugerían una deformación hacia el exterior. Este comportamiento probablemente venga motivado por querer reproducir deformaciones en cavidades, pero en el caso de los órganos —que sufren deformaciones hacia el interior y son el caso más abundante en una simulación— generaba colisiones en el sentido contrario al esperado. Por ello se decidió invertir este comportamiento, hecho que obliga a invertir las normales en objetos que representen cavidades y así las deformaciones se produzcan en el sentido correcto.

En las líneas futuras del proyecto se trata este aspecto y se proponen métodos para su mejora.

Tamaño de las mallas

En principio, los tamaños de las mallas asignadas a cada modelo de un elemento son independientes. Sin embargo, para que la simulación se ejecute de la manera más fluida y eficiente posible y para que el resultado sea óptimo se recomiendan las siguientes pautas:

- **Modelo de visualización.** Este modelo es el que el usuario verá en la pantalla y, por lo tanto, conviene aplicar aquí la densidad de puntos que se considere suficiente para la representación visual del objeto. Ésta debería ser la malla con mayor resolución con respecto a los otros dos modelos.
- **Modelo de colisión.** La malla de este modelo es invisible al usuario, pero entra en juego en el cálculo de colisiones. Este cálculo es muy costoso¹ y por ello conviene utilizar mallas de la menor resolución posible. La malla en cuestión debería ser la de menor resolución y no mayor que la malla de deformación, en cualquier caso.

¹El coste computacional del cálculo de colisiones depende del número de puntos de las mallas colisionantes y es del orden $O(n^2)$.

- **Modelo de deformación.** Lo ideal para la malla de este modelo es que su resolución sea igual o menor que la del modelo de visualización, a la vez que superior que la del modelo de colisión. Debe de tener el número de puntos suficientes para que las deformaciones se trasladen correctamente al modelo de visualización sin incurrir en artefactos visuales. Se detectó, sin embargo, que cuando la resolución de la malla de deformación igualaba a la de la malla de visualización, entonces el modelo de deformación parecía no responder al comportamiento de deformación aplicado (masa-muelle[21], por ejemplo). Este es un bug que no pudo ser localizado y corregido. Se desaconseja, por lo tanto, una resolución de malla igual a la del modelo de visualización.

5.2.4. Módulo de biomecánica

El cometido principal de la clase *vtkCollisionDetection*, como se vio en el apartado 3.3.6, es el de mantener un registro de las colisiones que tienen lugar en cada ciclo de simulación. No obstante, esta clase define dos métodos virtuales: *Initialize* y *Update*. Estos métodos han de ser definidos por la clase heredada, que es la que se encargará de realizar el cómputo de las colisiones.

En este caso, es la clase *vtkBioEngInterface* la que hereda de *vtkCollisionDetection* e implementa estos métodos. Concretamente, es el método *Update* el que realiza el ciclo de cálculo de colisiones correspondiente a cada ciclo de simulación. Es en este ciclo donde se detectó que las colisiones no se añadían al registro heredado de *vtkCollisionDetection*, lo que originaba que el método *GetCollisions* retornara siempre un conjunto de colisiones vacío.

Con los cambios introducidos, ahora las colisiones detectadas en cada ciclo de simulación son recopiladas en el atributo *Collisions* de tipo *vtkCollisionCollection*. Este conjunto de colisiones es vaciado previo a cada ciclo y vuelto a formar con las colisiones detectadas en el mismo.

Optimizaciones

La clase *vtkBioEngInterface* implementa en el método *Update* el cálculo de colisiones. En él recorre el conjunto de modelos implicados en el proceso de colisión mientras calcula las colisiones detectadas por cada par de modelos en dicho conjunto.

Inicialmente, este cálculo se realizaba con un bucle tal que el de la figura 5.1. Como puede verse, este bucle es de orden $O(n^2)$, pues por cada modelo del conjunto recorre una vez el mismo conjunto comprobando que los modelos entre los que va a realizar el cálculo son distintos. Es un orden esperable dada la naturaleza combinatoria de la operación, pero representa una ineficiencia importante por dos razones:

```

para i en 0 .. n-1 hacer
    m0 = modelos[i]
    para j en 0 .. n-1 hacer
        m1 = modelos[j]
        si m0 != m1 hacer
            Cálculo de colisiones entre m0 y m1
        fin si
    fin para
fin para

```

Figura 5.1: Bucle de cálculo de colisiones antes de aplicar las optimizaciones.

1. El cálculo de colisiones entre dos modelos es, como se explicará con más detalle en la sección 5.2.3, un cálculo de elevado coste computacional. Cualquier cálculo de más que se realice influirá notablemente en la eficiencia total de la simulación.
2. En este caso, la ineficiencia va más allá de la manera de recorrer el conjunto, pues se puede ver que cada pareja de modelos se computa dos veces. El cálculo de colisiones es independiente del orden, pero aquí se trata como si lo fuera y eso conlleva a doblar el número de cálculos necesarios. La operación relevante —la búsqueda de colisiones entre dos modelos— se realiza en este caso exactamente $n^2 - n$ veces.

En base a estas apreciaciones se concluyó aplicar una serie de optimizaciones básicas al bucle para corregir estos dos aspectos, que son la pauta para recorrer el conjunto y la independencia del orden del cálculo, obteniéndose entonces un código similar al de la figura 5.2.

```

para i en 0 .. n-2 hacer
    m0 = modelos[i]
    para j en i+1 .. n-1 hacer
        m1 = modelos[j]
        Cálculo de colisiones entre m0 y m1
    fin para
fin para

```

Figura 5.2: Bucle de cálculo de colisiones tras aplicar las optimizaciones.

A simple vista se puede ver que ahora, por cada ejecución del bucle externo, no se recorren todos los elementos de nuevo en el bucle interno sino que sólo se recorren los estrictamente necesarios. Esto consigue además que se pueda prescindir de la condición que comprueba que los dos modelos no

sean el mismo. Pero veamos ahora cuántas veces se ejecuta la búsqueda de colisiones en este caso.

El número de ejecuciones del bucle externo es fácilmente calculable.

$$i \in [0..n - 2] \rightarrow n - 2 - 0 + 1 = n - 1 \text{ veces} \quad (5.1)$$

En el caso del bucle interno, el número de ejecuciones depende del valor de i .

$$i \in [i + 1..n - 1] \rightarrow n - 1 - (i + 1) + 1 = n - (i + 1) = n - i - 1 \text{ veces} \quad (5.2)$$

Ahora veamos el número de veces que se ejecuta en total el interior de este bucle interno.

$$\sum_{i=0}^{n-2} n - i - 1 = n - 1 + n - 2 + \dots + 2 + 1 = \sum_{k=1}^{n-1} k = \frac{(n-1)n}{2} = \frac{n^2 - n}{2} \text{ veces} \quad (5.3)$$

Como se puede apreciar, con las modificaciones realizadas se ha conseguido reducir el número de ejecuciones de la búsqueda de colisiones entre modelos —la operación de mayor costo computacional— a la mitad.

$$\frac{n^2 - n}{2} < n^2 - n \quad (5.4)$$

Con esto se consiguió una mejora notable del rendimiento del proceso, lo que tendría repercusiones positivas de cara a fases más avanzadas del proyecto donde se analizarían los aspectos de fluidez e interacción.

5.2.5. Módulo de interacción

Al comienzo de este proyecto vtkESQui incluía un estilo de interacción por defecto, que es la clase *vtkDefaultInteractorStyle*. Este estilo hereda del estilo *vtkInteractorStyleTrackballCamera* y está diseñado para simulaciones de tipo laparoscópico, donde existe un canal para la cámara y otro u otros canales independientes para el instrumental. Era, por lo tanto, inválido de cara al nuevo tipo de simulación que se plantea en este proyecto.

En cualquier caso, se decidió tomar este estilo como base para la creación del estilo orientado a las simulaciones de endoscopia rígida monocanal, pero primeramente se realizaron una serie de restructuraciones en el código de *vtkDefaultInteractorStyle*. Dichas restructuraciones fueron necesarias para mejorar la legibilidad del código y para garantizar un buen funcionamiento del mismo, ya que los cambios fueron mayoritariamente en las estructuras condicionales, evitando así comportamientos imprevistos.

Además se decidió que, con vistas a facilitar su uso, se eliminaría la obligación de inicialización de la clase, sustituyendo esta particularidad por un flag que indicaría si la clase ha sido ya inicializada. Y en caso de que no lo haya sido, se realizaría la inicialización automáticamente.

Estilo de interacción de endoscopia rígida monocanal

Como se explicó en el apartado 2.1.1, en una intervención mediante endoscopia rígida monocanal se utiliza un canal único de acceso, a través del cual se introduce el endoscopio —que dispone de una lente y una luz incorporadas. El endoscopio utilizado en estos casos también proporciona un canal por el cual puede introducirse el instrumental necesario.



Figura 5.3: Imagen de un endoscopio utilizado para técnicas de histeroscopia. Se puede apreciar el cauterizador sobresaliendo en la punta del mismo.

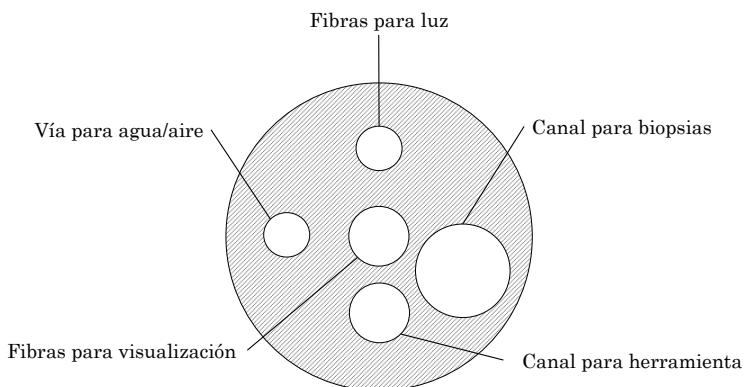


Figura 5.4: Ejemplo de sección transversal de lo que podría ser un endoscopio común.

Partiendo de este punto se creó la clase *vtkSingleChannelInteractorStyle*, que da soporte para la interacción con la escena de manera que se simule el comportamiento de un endoscopio rígido monocanal.

Actualmente, los estilos de interacción en `vtkESQuí` están diseñados para permitir la interacción a través de teclado y ratón. Así se proporciona al programador un modo de interacción con la simulación sin necesidad de disponer de un dispositivo periférico especial, normalmente a efectos de depuración, pero también podría mantenerse de cara al usuario como método alternativo de interacción. Así, esta nueva clase sobrecarga los métodos que se activan para cada evento de ratón y teclado, realizando para cada uno de ellos los cálculos y alteraciones en la simulación correspondientes.

Además, debido a que se simula también que el endoscopio dispone de una lente con cierto grado de inclinación, se implementan métodos *Set* y *Get* adicionales para configurar este parámetro.

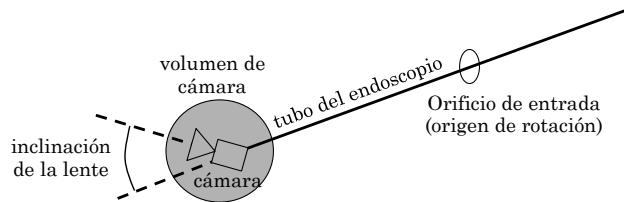


Figura 5.5: Mecánica de la simulación virtual del endoscopio rígido

Como puede verse en la figura 5.5, la mecánica del movimiento simulado es bastante diferente de la del estilo de interacción `vtkDefaultInteractorStyle`. Aquí el movimiento del endoscopio no sólo traslada y rota la herramienta, sino que hace lo mismo con la cámara virtual, pues el endoscopio porta la lente a través de la que se visualiza la escena. Como esta lente también puede tener una inclinación, la cámara virtual no tiene por qué estar orientada en la misma dirección que el endoscopio, sino que puede estar inclinada un cierto ángulo que coincidiría con el grado de inclinación de la lente.

Cámara virtual, volumen de cámara e instrumental han de moverse solidarios. Esto se consigue realizando las transformaciones con respecto al mismo origen. En concreto, para la cámara de VTK hay que prestar especial atención, pues hay que simular la inclinación de la lente y la interfaz de `vtkCamera` está enfocada a las transformaciones típicas de cámara: *Roll*, *Pitch*, *Yaw*, *Dolly*, *Azimuth*, *Elevation*²... Las transformaciones a emplear en este caso son de mayor complejidad y se requieren operaciones adicionales o la aplicación de distintas transformaciones básicas consecutivas.

Toda esta mecánica se tiene en cuenta para el movimiento de la cámara virtual y las herramientas virtuales a partir de los eventos de teclado y ratón

²Estas tres últimas son coordenadas radiales de uso típico en el ámbito de las cámaras. Corresponden a la distancia con respecto el centro, el radio horizontal y el radio vertical.

que recibe el interactor. También puede apreciarse en la figura 5.5 que se asigna un volumen a la cámara virtual para así poder calcular colisiones durante la navegación. Este objeto representativo del volumen de cámara es opcional y es identificado en tiempo de inicialización por el mismo estilo de interacción.

En la realidad, cuando se desea cambiar el instrumental utilizado durante una endoscopia de este tipo, es necesario extraer el endoscopio para montar la nueva herramienta, pero en el caso simulado se decidió permitir el cambio de instrumental sin necesidad de extraer el endoscopio. Tan sólo se requiere que la herramienta esté retraída antes de efectuar el cambio.

5.2.6. Módulo de soporte de dispositivos periféricos

Los dispositivos periféricos implementados en vtkESQui eran los dispositivos IHP y VSP de *Xitact* y los LSW de *Immersion*. En este proyecto se amplía el soporte para permitir la utilización de los dispositivos *Simball 4D* (SBM³) de *G-coder Systems*.



Figura 5.6: Conjunto de dispositivos periféricos simball. A la izquierda, modelo Simball 4D.

Con este fin se implementó la clase *vtkSBM*, que hace uso de los controladores del *Simball 4D* de *G-coder Systems* para finalmente proporcionar una interfaz independiente y adaptada a las necesidades de vtkESQui.

Uno de los cometidos de esta interfaz es adaptar los datos obtenidos del dispositivo periférico al entorno vtkESQui. Para ello, a menudo es necesario convertir el sistema de coordenadas y la magnitud de las medidas e incluso extraer cierta información implícita a partir de determinados datos en bruto o de distinta naturaleza.

En el caso del dispositivo *Simball 4D*, éste proporciona la inserción del instrumento y tres ángulos para determinar la orientación del mismo. Son los

³SimBall Medical

siguientes:

- Theta (θ). Ángulo desde el frontal del dispositivo hasta el orificio de entrada del instrumental.
- Phi (ϕ). Ángulo desde el lateral derecho del dispositivo hasta el orificio de entrada del instrumental.
- Gamma (γ). Ángulo de rotación en sentido contrario a las agujas del reloj sobre el eje del instrumental.

Todos los ángulos vienen dados en radianes, mientras que en vtkESQui se trabaja con grados. Además, como se aprecia en la figura 5.7, las referencias de los ángulos no se corresponden con las que requería la interfaz, así que a cada ángulo se le aplican las siguientes transformaciones antes de la conversión a grados:

- Theta. Corresponde al Pitch o cabeceo. En el dispositivo la referencia es el frontal del mismo, mientras que en vtkESQui la referencia se toma cuando el instrumental está centrado, así que a este valor se le resta $\frac{\pi}{2}$.
- Phi. Corresponde al Yaw o guiñada. En vtkESQui la referencia es el ángulo en que el instrumental está centrado y, además, el ángulo se mide en sentido opuesto. Es por esto que al ángulo se le resta $\frac{\pi}{2}$ y se invierte el resultado.
- Gamma. Equivalente al Roll o alabeo. Este valor se corresponde con los estándares de vtkESQui y por lo tanto no se ve alterado.

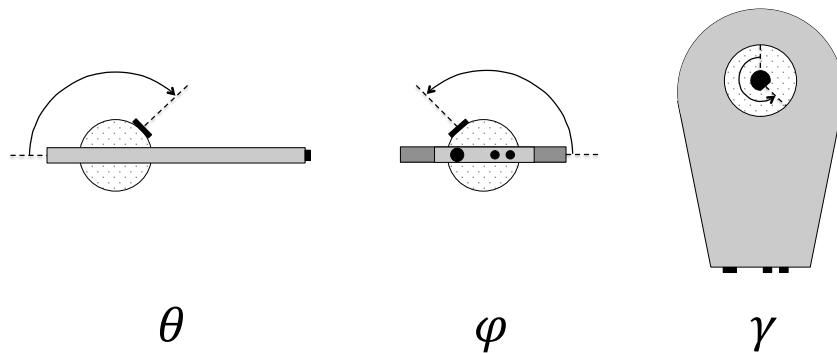


Figura 5.7: Coordenadas del sistema *Simball 4D*.

La clase *vtkSBM* también dispone de métodos para obtener información acerca de los instrumentos insertados y los pedales. Estos valores también son normalizados para cumplir con los estándares de vtkESQui, si bien su conversión es trivial, a diferencia de los ángulos tratados anteriormente.

Con motivo de la introducción del nuevo tipo de herramienta virtual de endoscopia rígida monocanal y para dar mejor soporte a todos los modos de interacción se decidió incluir métodos para poder especificar, previo a la inicialización, qué tipo de interacción se desea. Se trata de métodos *Set* y *Get* sobre la variable *SingleChannel*. Así se puede determinar si los movimientos del dispositivo periférico se traducirán a la simulación siguiendo el esquema de tipología laparoscópica o, por el contrario y si la variable tiene valor verdadero, según la mecánica del endoscopio rígido monocanal. A tal efecto se proporcionan también métodos para asignar una cámara de VTK, pues es necesario para simular la lente en este último caso.

El modo de endoscopia rígida monocanal tiene la peculiaridad de que sólo maneja una herramienta, a diferencia del modo de laparoscopia en el que se pueden asignar varias herramientas al dispositivo periférico.

5.2.7. Módulo de entrada/salida

Como se comenta en la sección 2.3.3, *vtkESQui* es capaz de tomar como base un documento SRML con el fin de reconstruir una simulación en particular, definida por los parámetros contenidos en dicho documento.

Anteriormente, esta labor se llevaba a cabo mediante analizadores sintácticos escritos en los lenguajes interpretados que se utilizan en *vtkESQui* (Tcl y Python). Existían pues dos analizadores distintos, que utilizaban distintos métodos para reconstruir la simulación, que interpretaban de manera distinta los datos del documento SRML y que estaban escritos en lenguajes distintos.

Esta duplicidad y heterogeneidad era una fuente importante de errores y confusiones, aparte de que no cumplía con los criterios de buena praxis programática, así que se decidió integrar el lector de SRML en las librerías de *vtkESQui*. Se creó para ello la clase *vtkSRMLReader*, la cual podría utilizarse a partir de ahora tanto desde C++ de manera nativa como desde Tcl o Python en su versión wrapeada, siendo en cualquier caso la misma clase con el mismo comportamiento.

La clase *vtkSRMLReader* hereda de la clase de VTK *vtkXMLDataParser*, cuyo cometido es el de leer un documento XML y facilitar su lectura mediante la creación de una estructura en memoria con la ayuda de la clase *vtkXMLElement*. Esta clase representa un elemento XML y todos los elementos anidados en él y proporciona métodos que facilitan la lectura de sus atributos, búsquedas y otras operaciones comunes.

Como colofón, se aprovechó la ocasión para realizar una implementación del lector consistente y versátil, estructurando el código de manera eficaz y legible y dotando a la clase de nuevos métodos para mejorar su funcionalidad

y facilitar su uso. Se implementó un único método público llamado *ConstructSimulation* que se encarga de todo el proceso de reconstrucción del objeto *vtkSimulation* en las siguientes fases:

1. Comprobación de la integridad XML del documento.
2. Lectura del documento SRML y construcción de la estructura *vtkXMLElement* conteniendo la raíz del mismo.
3. Comprobación de que la estructura del documento coincide con la especificación SRML actual de vtkESQui y de que los atributos son correctos.
4. Construcción del objeto *vtkSimulation* y retorno del puntero al mismo.

En caso de detectarse algún error o de fallar alguna comprobación, se comunica mediante un error de VTK y devolviendo un puntero nulo.

De esta forma, empleando la clase *vtkSRMLReader* es posible tanto obtener parámetros específicos del documento SRML a través de los métodos heredados, como de comprobar la construcción del documento y reconstruir un objeto *vtkSimulation* de una manera sencilla gracias a los nuevos métodos implementados. El método público *ConstructSimulation* realiza las operaciones comentadas anteriormente de manera automática y, si todo procede correctamente, devuelve un puntero a un objeto *vtkSimulation* con los componentes y la configuración especificados en el documento SRML.

Durante la implementación y prueba de la clase *vtkSRMLReader* se detectó que el tipo de herramienta virtual *vtkToolEndovascular* carecía de medios para identificar el modelo de herramienta. Puesto que la solución era trivial, este defecto se corrigió de cara a futuras implementaciones.

5.2.8. Casos particulares en los distintos sistemas operativos

Si bien todo el entorno ESQUI está enfocado desde un principio para facilitar el desarrollo multiplataforma, es inevitable encontrar ciertas dificultades de adaptación y migración durante el desarrollo. Concretamente, en este proyecto, la mayor parte del desarrollo se llevó a cabo en sistemas UNIX. A continuación se detallan las diversas adversidades que se encontraron.

Mac OS X

Al comienzo del proyecto se trató de ejecutar ejemplos básicos de vtkESQui en las distintas plataformas. Fue entonces cuando se detectó un bug en *Mac OS X* que impedía la ejecución de la mayoría de ellos, terminando la ejecución del programa de manera brusca y con un fallo de página.

La localización del bug fue en un principio infructuosa, de manera que se centró el desarrollo en las plataformas restantes. Fue más adelante, durante la

inspección de la clase *vtkSyncPolyDataFilter*, donde se detectó su origen. La clase en cuestión se encarga de leer una malla de un fichero *vtp* y de sincronizar los puntos con los de una malla adicional proporcionada a tal efecto.

El origen del bug estaba en el constructor de la clase *vtkSyncPolyDataFilter* y es cuanto menos curioso: al flag de inicialización —que se emplea para controlar si la clase ha sido inicializada o no— no se le daba un valor inicial. Cabe pensar que esto no debería causar ningún problema, si bien es mala práctica y probablemente fue un simple fallo humano no tenerlo en cuenta, pero si no se especifica un valor inicial éste puede ser cualquiera.

Se comprobó que en *Mac OS X Snow Leopard (versión 10.6.8)* el valor por defecto del tipo booleano de C/C++ es verdadero. Esto ocasionaba que la instancia del *vtkSyncPolyDataFilter* se diera por inicializada cuando no lo estaba y se realizaran accesos a memoria inapropiados, lo que provocaba finalmente el fallo de página y el aborto de la ejecución.

Microsoft Windows

La adaptación a los sistemas Microsoft Windows se realizó en un estado relativamente avanzado del proyecto, con vistas a depurar y mejorar el nuevo soporte periférico que se había implementado. Este hecho, sumado a que era la primera vez que se incorporaba el módulo de soporte periférico a la compilación desde el comienzo del proyecto, supuso la aparición de numerosas incompatibilidades —y a menudo difíciles de resolver.

Durante este proceso de depuración se descubrió una deficiencia en el código de especificación del módulo de interacción.

En *vtkESQui*, cada módulo se encuentra en un directorio distinto y cada uno de ellos contiene un fichero llamado *vtkESQui<módulo>Win32Header.h*— donde *<módulo>* es el nombre del módulo correspondiente —que incluye el código necesario para realizar la importación y exportación de las librerías DLL en Windows.

El módulo de interacción era el más reciente de la librería y la clase *vtkDefaultInteractorStyle* estaba incluyendo el archivo de otro módulo, ya que el fichero correspondiente al módulo de interacción no existía. Esto era lo que estaba generando el problema y se solucionó creando dicho fichero e importándolo correctamente en el código de *vtkDefaultInteractorStyle*, así como en las otras clases que posteriormente se implementaron en dicho módulo.

Además, se apreció que las librerías dinámicas de los controladores del dispositivo periférico han de ser copiadas en el directorio de librerías dinámicas de *vtkESQui* o, en su defecto, incluir el directorio donde éstas se encuentran en la variable \$PATH del sistema.

5.3. Código en Python

Como se comentó en la sección 3.3.6, la librería vtkESQui proporciona una interfaz en varios lenguajes de scripting. Esto permite un uso más dinámico de la misma, así como la integración con interfaces gráficas administradas a dicho nivel. En este proyecto se aprovecha que la librería está wrapeada en Python para desarrollar así el software de manera más dinámica y ágil.

Durante este proceso se trataron varios aspectos que se comentan en los siguientes apartados.

5.3.1. Referencias en Python

La versión inicial de vtkESQui había sido probada principalmente utilizando Tcl, mientras que en lo referente a Python simplemente se mantuvo como puerta abierta a futuras implementaciones. Partiendo de este punto, es necesario apuntar que hay que tener en cuenta ciertos defectos del wrapeado en vtkESQui. Dichos defectos ocasionaban que los objetos conservaran referencias a otros objetos, pero que dichas referencias quedaran huérfanas al eliminar el gestor de memoria de Python dicho objeto puesto que las referencias en cuestión no estaban contabilizadas.

Soluciones planteadas

A primera vista, el error era ocasionado por algún defecto en el wrapeo de las funciones. De hecho, en la wiki de VTK[22] se hace referencia a la problemática que representa en varios casos la adaptación de cierto tipo de funciones y métodos para su ejecución sobre Python. Además, en el código fuente de VTK se incluye `README_WRAP.txt`, que advierte de los métodos que no serán wrapeados:

- Métodos cuya lista de parámetros incluya punteros a objetos de tipos distintos a `vtkObject`, `char` o `void`. Una excepción son los métodos de la clase `vtkdataArray`.
- Métodos que devuelvan un puntero a un tipo distinto de `vtkObject`, `char` o `void`, a no ser que el método disponga una entrada en el fichero `hints` (pista para el wrapeo). Los métodos de `vtkdataArray` son nuevamente una excepción.
- Métodos cuya lista de parámetros incluya un tipo enum.
- Métodos que sean operadores, aunque existen varias excepciones.

Sin embargo, este error debía de ser causado por otra razón, pues todos los objetos de vtkESQui heredan de `vtkObject`. Así, una posible vía de acción era investigar más a fondo el mecanismo de wrapeo de VTK y observar la

aplicación de las pistas de wrapeo (*hints*).

Otra posibilidad era mantener las referencias problemáticas en ámbito, de manera que Python las tuviera contabilizadas y el recolector de basura no eliminara el objeto.

Solución adoptada

Lamentablemente, y debido a que hubo que acotar el trabajo y a que este aspecto en particular requería bastante dedicación, no pudo dársele la solución óptima a esta problemática. Se decidió mantener las referencias en ámbito explícitamente para evitar tener que revisar y probablemente actualizar el código de wrapeo de VTK, lo que habría disparado la complejidad del proyecto.

5.3.2. Integración VTK-wxPython

Para poder dotar a la aplicación de una interfaz gráfica que permita al usuario interactuar con la aplicación de una manera cómoda y sencilla es necesario integrar el código de VTK con el de las librerías gráficas —en este caso wxPython. Para ello se consideraron los siguientes aspectos.

Inclusión de una ventana VTK en la interfaz gráfica

VTK dispone de dos clases distintas en Python que permiten empotrar una ventana de renderizado VTK en una interfaz gráfica wxPython: *wxVTKRenderWindow* y *wxVTKRenderWindowInteractor*. A pesar de la diferencia clara en el nombre, a efectos prácticos la diferencia es más sutil y confusa. Si a esto le sumamos que escasea la documentación sobre las mismas, su empleo se complica ligeramente. Las siguientes líneas han sido escritas pues en base al comportamiento observado de las clases y la experiencia en su utilización.

La clase *wxVTKRenderWindow* proporciona una funcionalidad bastante reducida. Parece estar orientada a mostrar escenas extremadamente simples y no permite el uso de estilos de interacción ni emplea Z-buffer para el renderizado. Por otro lado, la clase *wxVTKRenderWindowInteractor*, a pesar de lo que su nombre pueda indicar, no es un wrapping de la clase *vtkRenderWindowInteractor* —en cuyo caso podría complementar a la clase *wxVTKRenderWindow* para suministrar la funcionalidad faltante— sino que se trata de una ventana de renderizado VTK que sí utiliza Z-buffer y que provee de funcionalidad adicional para que se comporte de manera similar a un interactor genérico de VTK. A efectos prácticos, pudiera decirse que esta última clase es en cierta forma una herencia múltiple entre *vtkRenderWindow* y *vtkGenericRenderWindowInteractor*.

Esta tendencia a la confusión originada por los nombres de las clases hizo que, en principio, se decidiera utilizar la clase *wxVTKRenderWindow* y que se desechara el uso de la otra, con lo que se encontraron diversos problemas de renderizado (superposición errónea de objetos a causa de la falta de Z-buffer) y de interacción (porque no existía tal funcionalidad). Finalmente se descubrió el propósito real de la clase *wxVTKRenderWindowInteractor* y se dio solución a dichos problemas.

Se trata de versiones muy recientes y la funcionalidad no es igual de completa que con el uso directo de una ventana VTK, requiriendo incluso que haya que completarlas ligeramente para su posterior uso, pero el funcionamiento es correcto en la gran mayoría de casos.

Mejoras en la clase escogida

Esta clase está implementada en Python y es proporcionada por la librería VTK. Se encuentra en un estado de desarrollo bastante temprano y proporciona una funcionalidad básica de cara a la incorporación de una ventana VTK en una interfaz gráfica de usuario de wxPython.

Un ejemplo de esto es que la clase, por defecto, no da soporte a todos los símbolos del teclado con los que trabaja VTK. Esto motivó la creación de una clase —que se decidió llamar *SimulationRWI*— que sobrecargara el método *OnKeyDown*, encargado de atender los eventos de presionado de tecla. Así pudo dársele soporte a teclas adicionales como son las teclas direccionales y teclas de avance y retroceso de página. También se aprovechó la ocasión para restringir las teclas que se manejarían a las imprescindibles para la interacción diseñada, evitando así que se desencadenen acciones no deseadas.

Además se sobrecargó el método *OnSize*, que es llamado cada vez que la ventana es redimensionada, para así notificar al objeto *vtkScenario* del cambio de tamaño de ventana y que el renderizado discurra de manera correcta. De no hacerse así, la imagen renderizada podría resultar descentrada o con un factor de ampliación incorrecto.

Como se puede apreciar en la figura 3.3, el pipeline de VTK dirige, a través de un objeto renderizador, la información de la escena sobre una ventana VTK. Con la funcionalidad de renderizado cubierta y los filtros de teclas aplicándose correctamente, esta ventana es capaz de integrarse en la interfaz de *wxPython* a la perfección. A través de ella, el usuario visualiza la escena simulada y, controlando las entradas del renderizador, el contenido de la ventana es fácilmente alterable.

Ventana huérfana

El elemento VTK empotrable se trata de una ventana VTK (*vtkRenderWindow*) integrada en la interfaz de wxWidgets como hija de la ventana principal y, por

lo tanto, formando parte de ella.

Inicialmente se daba el caso de que se creaba automáticamente una ventana adicional, huérfana e independiente de la interfaz de wxWidgets. Posteriormente se detectó que esto ocurría como consecuencia de la inicialización del interactor previo al comienzo de la ejecución del bucle principal de la aplicación. Durante la inicialización, el interactor trataba de llevar a cabo un primer renderizado en la ventana VTK que no se encontraba aún mapeada en la ventana principal de la aplicación.

Este comportamiento se solucionó postergando la inicialización del interactor hasta después del comienzo del bucle principal de la aplicación.

Método de control del proceso de simulación y renderizado

La metodología habitual de funcionamiento de VTK es mediante el método *Run* del interactor asociado a la ventana. Dicho método es un bucle continuo de espera activa en el que se lanzan y capturan los eventos pertinentes.

Esta metodología choca frontalmente con la de wxPython, pues en una aplicación wxPython debería coexistir con el bucle de aplicación, donde se tratan todos los eventos de aplicación. Ambos no pueden ejecutarse simultáneamente en el mismo hilo de ejecución porque el primero que se ejecute lo absorberá en un bucle. Para acoplar estas dos partes se plantearon las siguientes soluciones:

Ejecutar bucle de VTK en un hilo paralelo

Esta era una opción trivial y por ello la primera en plantearse. Según este planteamiento, la aplicación se ejecutaría y, una vez estuviera en marcha el bucle principal de la aplicación y se cargara la simulación, el bucle del interactor de VTK se lanzaría en un hilo diferente. En la teoría, esto significaría que ambos bucles no se interbloquearían y que, salvo los interbloqueos de memoria que gestionaría Python de forma nativa, ambos serían independientes y la aplicación se ejecutaría de manera fluida.

En la práctica, el hecho de que el bucle del interactor de VTK se ejecutara mediante espera activa y con una tasa de repetición elevada significó que este hilo acabó por acaparar el tiempo de ejecución, la aplicación en sí veía bloqueada su interfaz gráfica y lo único que se ejecutaba de manera relativamente fluida era la ventana incrustada de VTK.

Utilizar temporizadores de wxPython

Como alternativa, es posible utilizar un temporizador *wxTimer* de activación continua para marcar los ciclos de renderizado y prescindir del bucle del interactor de VTK. Así se evita que el interactor acapare la mayoría del tiempo de ejecución y se mantiene receptiva la interfaz gráfica de la aplicación mientras

la simulación virtual se renderiza.

Cuando se ejecuta la simulación de manera independiente, utilizando el bucle del interactor de VTK, lo normal es emplear un ciclo de renderizado de entre 1 y 5 milisegundos. Sin embargo, los temporizadores *wxTimer*, así como ocurre con los temporizadores propios de Python, tienen una precisión que no va más allá de los 15 milisegundos. Esto impide que mediante este método se puedan emplear tasas de simulación por encima de 66 ciclos por segundo.

No obstante, y gracias a las optimizaciones que se comentan en el apartado 5.2.1, es suficiente con emplear una tasa de simulación en torno a 25 ciclos por segundo, estando ligado este parámetro únicamente a criterios de visualización con el objetivo de que el usuario pueda interactuar de manera fluida y que los cambios en la imagen visualizada también lo sean.

Esta aproximación fue la escogida como definitiva y se fijó una tasa de simulación de 25 ciclos por segundo. Las razones para escoger esta tasa de simulación se comentan en el apartado 5.2.4.

5.3.3. Visualización de colisiones

De cara a la producción de la aplicación final y puesto que, como se comenta en el apartado 3.2, no se dispondrá de feedback táctil, es conveniente proveer al usuario de la aplicación de algún tipo de feedback visual. A continuación se explican las distintas opciones que se barajaron y los resultados que se obtuvieron en cada caso.

Método de visualización por defecto

Previo al comienzo de este proyecto, la librería vtkESQuí contemplaba la posibilidad de resaltar las colisiones de los objetos mediante el método de activación *SetDisplayCollisions* de la clase *vtkCollisionModel*. Cuando se activaba este procedimiento, se alteraban los colores de la representación visual del modelo de colisión para que los puntos de dicho modelo que hubieran colisionado se tornaran de color azul, mientras que el resto fueran de color blanco.

Este procedimiento se desechó desde el principio, puesto que de cara a la simulación se busca cierto nivel de realismo y, como factor adicional a tener en cuenta, la representación del modelo de colisión podría confundir al usuario y desviar su atención del ejercicio simulado.

Alteración de la textura

Para conseguir este objetivo se buscaba un efecto de enrojecimiento, dando a entender que las colisiones resaltadas producen algún tipo de daño o irritación en los tejidos involucrados. En este caso se trató de alterar la textura del

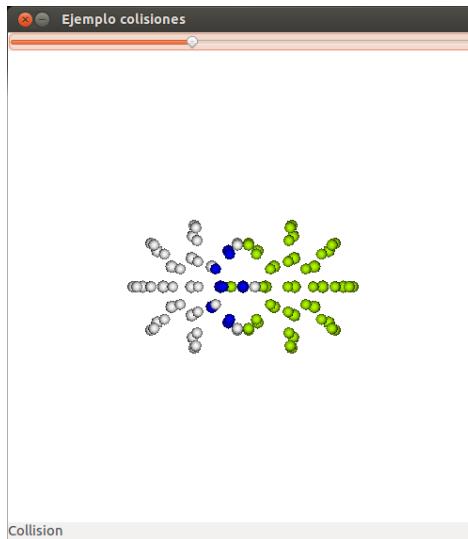


Figura 5.8: Visualización del modelo de colisiones mediante el uso de la clase *vtkGlyph3D*.

modelo de visualización para darle un color más rojizo.

VTK da la posibilidad de colorear un objeto mediante la asignación de escalares a los puntos de una malla y la utilización de una tabla de referencia (*lookup table*) para así asignar a cada escalar un color. Para realizar estas acciones, toda esta información se pone a disposición del mapper, cuya función principal es la de convertir las mallas que se van a renderizar en polígonos primitivos.

La idea era activar la visualización mediante un método llamado *SetVisualizeCollisions* en la clase *vtkScenarioElement*. De esta manera, desgraciadamente, la información proporcionada al mapper entraba en conflicto con la información de textura y no podía apreciarse correctamente. A la vez, este conflicto provocaba la pérdida de la trama de la textura y sólo podía verse el color de fondo de la misma.

Posteriormente se contemplaron otras maneras de alterar la textura del modelo de visualización del objeto, pero fueron rápidamente desechadas, bien por la complejidad computacional del proceso, bien por la dificultad técnica del mismo.

Con polígonos adicionales

Finalmente se decidió realizar el procedimiento de resaltado de manera independiente a la librería *vtkESQui*, llevando a cabo todas las operaciones desde el nivel de script de la aplicación. La idea es crear una capa rojiza y semitransparente en la zona de colisión. Para ello, se localizarían en primer lugar

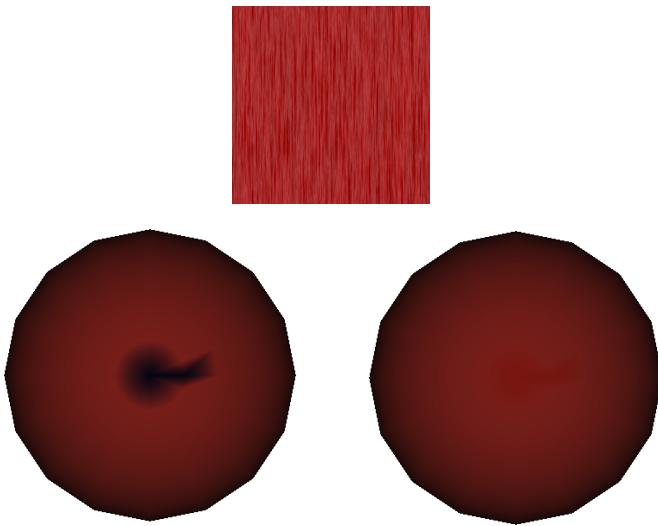


Figura 5.9: Arriba, textura original del objeto. Abajo, efectos del intento de alteración de la textura a través del mapper. Obsérvese cómo se pierde la textura en el proceso.

los elementos susceptibles de sufrir el resaltado y en cada ciclo de simulación se realizarían los siguientes pasos para cada elemento:

1. Comprobar si el elemento está habilitado, en caso contrario no se realiza el proceso. El procedimiento de visualización de colisiones tiene un coste computacional considerable y es conveniente aplicarlo el menor número de veces posible.
2. Obtener colisiones del elemento. De no haberlas se abortaría el proceso para el elemento en cuestión y se pasaría al siguiente.
3. Trasladar los puntos de colisión al modelo de visualización, así obtenemos una serie de puntos en el modelo de visualización que corresponden a los lugares donde tuvieron lugar las colisiones.
4. Obtener las celdas a las que pertenecen dichos puntos. Las celdas se corresponden con las caras del modelo de visualización. Esto es importante con vistas a duplicar dichas caras en la capa adicional que se pretende crear.
5. Ampliar la selección de caras con las celdas colindantes. Como el malla-doo del modelo de colisión es de baja resolución, obtener sólo las caras más cercanas a los puntos de colisión podría hacer que se resaltaran varias caras aisladas, provocando un efecto de moteado rojizo en lugar del enrojecimiento de la región que se pretende.
6. Duplicar las caras seleccionadas y asociarlas a un actor independiente, aplicando las propiedades de color y opacidad pertinentes.

- Aplicar un factor de escalado muy pequeño para que la nueva capa no se fusiona con el elemento original y se pueda visualizar sobre el mismo. Aquí se discrimina si el elemento representa una cavidad para aplicar en estos casos un escalado de factor menor a 1 y así la capa adicional no quede oculta tras la superficie del objeto.



Figura 5.10: Representación de una colisión en una esfera mediante duplicado de caras. Se puede apreciar que la capa flota sobre el objeto y que deja ver la textura del mismo.

5.3.4. Simulación de la visualización de la lente

En la imagen de una endoscopia rígida monocanal real, un aspecto característico es el contorno circular de la misma. Esto es debido a la forma de transmisión de la imagen, donde el canal a través del que se introducen las fibras es de sección circular. En esta imagen se aprecia además una muesca que indica la orientación de la lente. Así, cuando ésta tiene cierto grado de inclinación es más fácil conocer hacia dónde se está enfocando.

Esta característica se simula mediante un plano en forma de disco con muesca que se mantiene siempre perpendicular a la dirección de la cámara. La posición se mantiene gracias a la clase de VTK *vtkFollower* que realiza tal cometido.

5.3.5. Proceso de simulación y renderizado

A efectos de controlar el proceso de simulación y renderizado, así como el flujo de la aplicación y el dispositivo periférico opcional, el prototipo de aplicación cuenta con un temporizador de activación continua que administra todos los procesos involucrados.

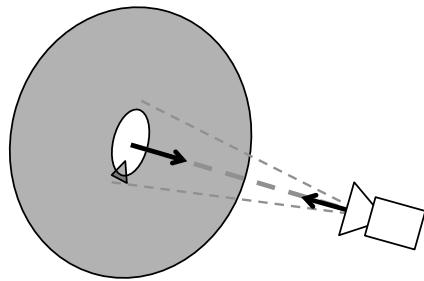


Figura 5.11: Esquema de las posiciones relativas entre cámara virtual y el disco con muesca.

Flujo de aplicación

El temporizador actúa como una máquina de estados que mantiene control del flujo de la aplicación cuando se utiliza con un dispositivo periférico *Simball 4D*. En tal caso, la interacción mediante teclado y ratón se desactiva y el *Simball 4D* se convierte en el único interacto. Mediante tal máquina de estados se controlan los siguientes estados de la interacción:

1. **Estado inicial.** Estado en el que se comienza justo tras la carga de la simulación. Se comprueba que la herramienta del dispositivo periférico se encuentre fuera del trócar. En caso contrario, se muestra un mensaje solicitando al usuario que la extraiga.
2. **Estado de selección de lente e instrumental.** Este es el estado que se mantiene cuando la herramienta del dispositivo periférico se encuentra fuera del trócar. Aquí se pueden realizar las siguientes acciones:
 - *Cambio de lente.* Con los pedales del dispositivo *Simball 4D* se puede cambiar la selección de lente.
 - *Cambio de herramienta.* Si se realiza la misma acción con las pinzas de la herramienta cerradas se puede cambiar la selección de la herramienta.
3. **Estado de simulación.** En este estado se ejecuta una iteración del bucle de simulación por cada activación del temporizador. Además, en caso de haber interacción háptica, los pedales controlan la inserción de la herramienta virtual.

Bucle de simulación

Este aspecto se controla también desde el temporizador. Por cada activación del mismo, encontrándose en el estado de simulación, se ejecuta una iteración del bucle. Dicho bucle realiza las siguientes operaciones en el siguiente orden:

1. **Interacción.** Da la orden al objeto del dispositivo periférico para que actualice los objetos del escenario. También se encarga de mantener actualizado el disco de la lente.
2. **Paso de simulación.** Ejecuta el método *Step* del objeto de simulación. Éste método realiza la mayoría de operaciones involucradas en la simulación, como la detección de colisiones.
3. **Corte de órganos.** En caso de estar seleccionada la herramienta de cuchilla, se controlan las colisiones de ésta para simular el corte del órgano con el que colisiona.
4. **Resaltado de colisiones.** En cada ciclo de activación, se recopilan las colisiones que se han detectado para resaltarlas visualmente con una capa rojiza superficial. Esta capa se crea recolectando las celdas que han colisionado, junto a las colindantes de vecindad directa, y duplicándolas. Luego se le asocia el color rojo al actor y se le aplica una opacidad del 50 %.
5. **Renderizado.** Finalmente, se da la orden a la ventana integrada de realizar el renderizado visual de la escena.

Conflicto con método *Update* en dispositivo periférico

Durante las pruebas con dispositivo periférico *Simball 4D* en sistemas Microsoft Windows se detectó un problema de sincronía entre los objetos controlados por el dispositivo periférico y el resto de la escena. La escena se actualizaba con la frecuencia que determinaba el temporizador de control de la simulación, pero la cámara y los objetos controlados por el dispositivo periférico lo hacían a una frecuencia muy superior.

Esto, a parte de ser indeseable por el simple hecho de que implicaba que una serie de acciones indeseadas se estaban llevando a cabo automáticamente, escapando al control del temporizador de la simulación, producía un efecto de temblor en el disco de la lente. Este efecto venía causado por la diferencia en la tasa de actualización entre el disco y la cámara, pareciendo que al disco *le costaba seguir* el movimiento de esta última.

Se descubrió, finalmente, que la actualización no controlada de esos elementos de la escena tenía origen en los mismos procesos de actualización del dispositivo periférico. El dispositivo periférico *Simball 4D* actualiza sus datos de posicionamiento 50 veces por segundo.

Capítulo 6

Prototipo de simulador de técnicas de endoscopia rígida monocanal

Se implementó un prototipo de aplicación en el que se ven reflejados todos los aspectos trabajados en el mismo. Particularmente, se decidió enfocarlo a las técnicas de histeroscopia y se le dió al prototipo el nombre de *HysTrainer* (Hysteroscopy Trainer, que significa entrenador de histeroscopia).



Figura 6.1: Logotipo de la aplicación prototipo con las siglas referentes al nombre. La forma del mismo se inspira en la imagen obtenida de un endoscopio rígido monocanal.

6.1. Funcionalidad

Este prototipo es un ejemplo de aplicación docente enfocado al entrenamiento de técnicas de histeroscopia. Fomenta el perfeccionamiento de estas técnicas mediante la utilización de un dispositivo periférico *Simball 4D*[3] y tres ejemplos, fundamentalmente centrados en el entrenamiento de las capacidades de navegación del usuario.

La aplicación permite al usuario cargar un ejemplo concreto a través de un archivo SRML y comenzar a entrenar sus habilidades. Durante el entrenamien-

to se alertará visualmente al usuario de colisiones indeseadas, resaltando en rojo la zona involucrada. Las características funcionales específicas que cumple el prototipo se especifican con mayor detalle en el apartado 3.1.

6.2. Flujo de la aplicación

El prototipo ofrece una interfaz gráfica sencilla, intuitiva y práctica, que pone a disposición del usuario las operaciones esenciales y necesarias para la carga de una simulación e interacción con la misma. En la figura 6.2 se presentan todos estos elementos.

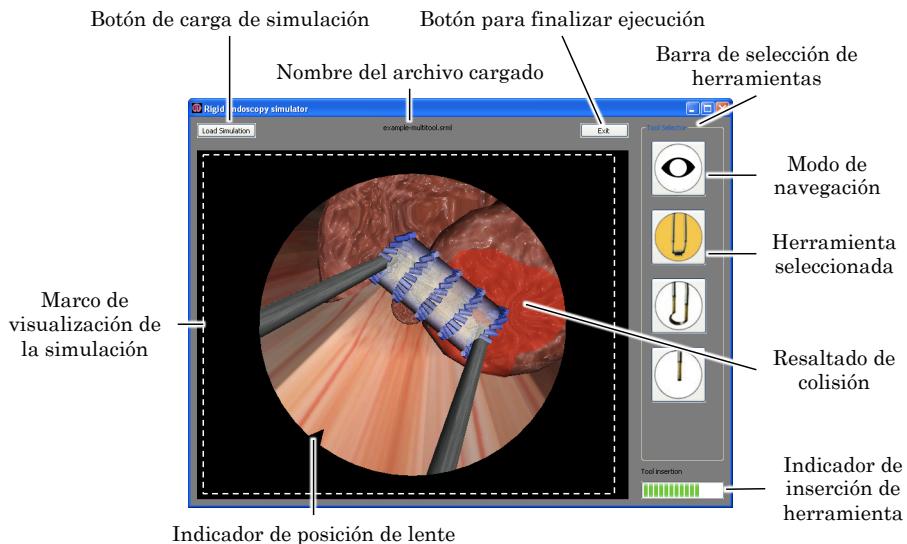


Figura 6.2: Elementos básicos de la interfaz gráfica del prototipo de simulador.

Existen también una serie de vistas adicionales que ayudan al flujo de la aplicación:

- **Vista inicial.** (Figura 6.3) Es el estado inicial de la aplicación. Aquí todos los elementos de la interfaz se encuentran desactivados, a excepción del que permite cargar una simulación y del botón de salida. También puede encontrarse esta vista tras algún error en la carga de una simulación.
- **Vista de selección de lente.** (Figura 6.4) Esta vista podemos encontrarla justo después de la carga de una simulación o cuando se extrae el instrumental del dispositivo periférico. Permite al usuario escoger la lente que desea utilizar en la simulación.
- **Vista de simulación.** (Figura 6.5) Es la vista principal de la aplicación, pues es donde se emplea la mayoría del tiempo de ejecución. Aquí se

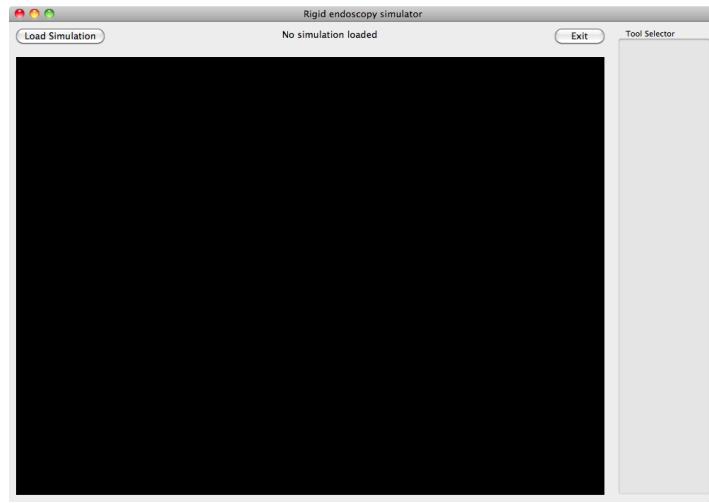


Figura 6.3: Vista inicial.

visualiza la simulación, a la vez que es posible escoger la herramienta que se desea utilizar. También permite cargar una nueva simulación, abortando así la que se encuentra en proceso.

6.3. Finalidad

Este prototipo de aplicación para la simulación de técnicas de histeroscopia se presenta como una muestra del potencial del entorno ESQUI para la ejecución de simulaciones útiles en la formación sanitaria. Además, ilustra la validez de los simuladores virtuales para el entrenamiento de técnicas sanitarias.

Si bien la aplicación, como prototipo, no cumple con todos los requisitos de una aplicación docente, es fundamental como base para la implementación de futuras aplicaciones docentes orientadas a las técnicas de histeroscopia o a cualquier tipo de técnica de endoscopia rígida.

La realización de este proyecto ha significado la consecución de una serie de objetivos, a la vez que supuesto la resolución de un conjunto de problemáticas y cuestiones de gran utilidad para futuras implementaciones de simuladores virtuales basados en la tecnología de ESQUI. Todo ello ha desembocado en la identificación de múltiples líneas futuras, orientadas no sólo a una aplicación docente de este prototipo, sino también a la mejora de la plataforma ESQUI y a su aplicación en otras técnicas sanitarias.

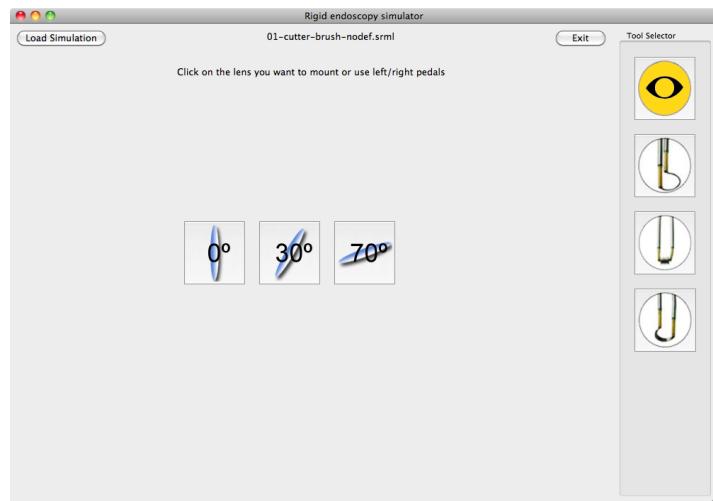


Figura 6.4: Vista de selección de lente.

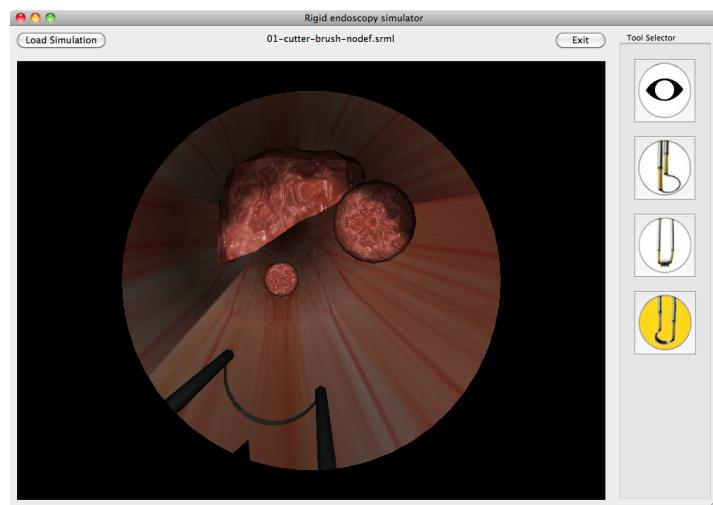


Figura 6.5: Vista de simulación.

Capítulo 7

Manual de usuario

En este manual de usuario se describe el proceso de instalación y de uso de *HysTrainer*, la aplicación prototipo producto de este proyecto.

7.1. Instalación

- **En sistemas UNIX.** (Linux o Mac OS X) La aplicación es autocontenido y no necesita instalación ninguna. Por comodidad, es recomendable que se copie la carpeta de la aplicación al directorio de aplicaciones del sistema.
- **En Microsoft Windows.** Antes de poder ejecutar la aplicación es necesario que la instale en el sistema. Simplemente ejecute el instalador (*HysTrainerInstaller.exe*) y siga las instrucciones. Al concluir la instalación, deberá tener una nueva entrada en los programas del menú de inicio. En dicha entrada tendrá acceso a la aplicación, la licencia y el asistente de desinstalación.

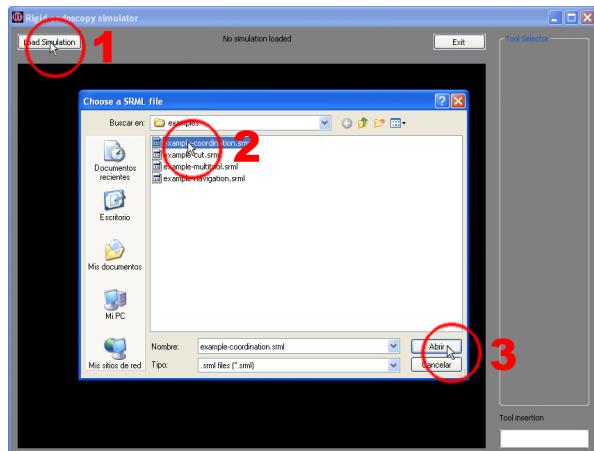
7.2. Guía de inicio

A continuación se describirán los pasos necesarios para ejecutar un ejemplo simple y así ver el funcionamiento general de la aplicación. La guía está orientada a la utilización con un dispositivo *Simball 4D*, pero también se adjuntarán los controles alternativos mediante ratón y teclado.

1. Ejecute la aplicación *HysTrainer*. Los usuarios de Microsoft Windows podrán encontrar el acceso directo a través del menú de inicio, mientras que en el resto de sistemas simplemente se tendrá que ejecutar la aplicación autocontenido.



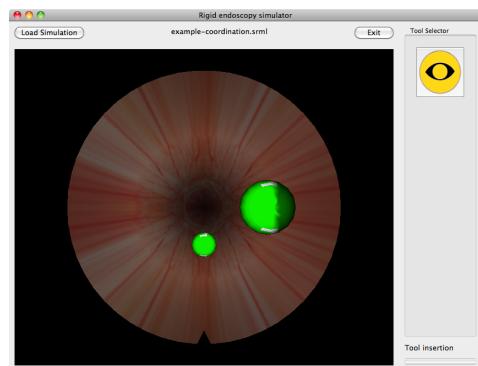
2. Presione el botón *Load Simulation*. Se abrirá un diálogo en el que se podrá seleccionar un archivo SRML para cargar. Por defecto, el diálogo muestra los SRML de ejemplo de la aplicación. Abra el ejemplo **example-coordination.srml**.



3. Ahora verá la vista de selección de lentes. Utilice los pedales derecho e izquierdo para mover la selección entre las distintas lentes. Seleccione la lente recta (de 0 grados). La simulación comenzará al introducir el dispositivo en el trócar.

Controles alternativos:

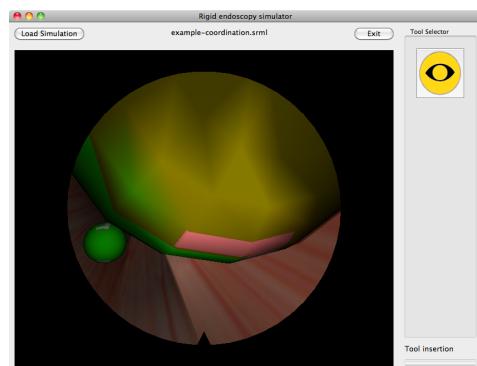
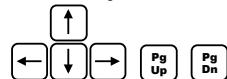
- ↳ Click sobre el botón de la lente deseada.



- Mueva el dispositivo *Simball 4D* para mover el histeroscopio virtual. Vea cómo puede también controlar la profundidad.

Controles alternativos:

↳ Click y arrastra. Rueda de desplazamiento para mover en profundidad.



Cuando la lente colisione con la esfera verá cómo aparece una sombra rojiza.

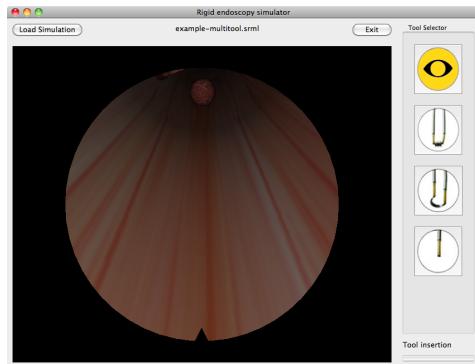
- Retire un poco el endoscopio hacia atrás. Si ahora rota el dispositivo *Simball 4D* sobre su propio eje podrá ver cómo se mueve la pestaña negra del borde. Eso significa que está rotando la lente del endoscopio, si bien no se aprecia nada más visualmente porque la lente es recta.

Controles alternativos:

↑ + ↗ Click y desplazamiento lateral.

z x

6. Siga ahora el mismo procedimiento del inicio para cargar un SRML. Cargaremos ahora el ejemplo `example-multitool.srml` y seleccionaremos la lente de 30°. La visión que tendrá ahora no será tan directa, sino que tendrá cierto grado de inclinación debido a la lente. Podrá comprobar que al rotar la lente su visión se orientará siempre hacia la pestaña del borde.



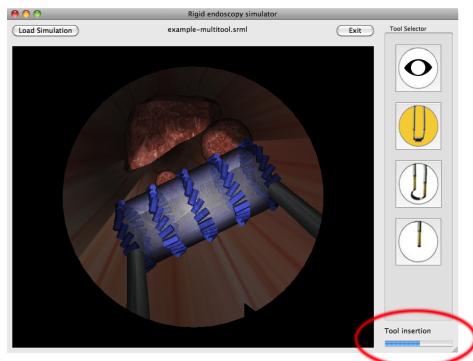
7. A la derecha podrá ver ahora una columna de botones que representan las herramientas disponibles. Resaltado en amarillo aparece el botón seleccionado, que inicialmente es el de visualización (no hay ninguna herramienta seleccionada). Para cambiar de herramienta extraeremos el dispositivo *Simball 4D* y mantenemos cerrada la pinza mientras usamos los pedales para cambiar la selección. Seleccionamos el cepillo —primera herramienta desde arriba— y volvemos a introducir el dispositivo. Una vez dentro, presione el pedal derecho hasta que el cepillo aparezca en pantalla. Observe que la barra de inserción comienza a llenarse a medida que la herramienta avanza.

Controles alternativos:

↗ Click sobre el botón de la herramienta para seleccionarla.

↑ + ↗ (Rueda de desplazamiento) Introducir o extraer herramienta.

↑ + Pg Dn Pg Up Introducir o extraer herramienta.

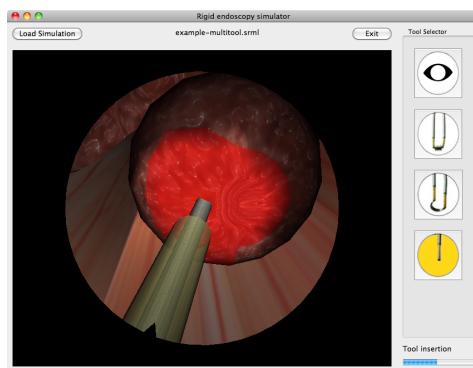


Al tocar los órganos con la herramienta verá cómo también se enrojece la zona.

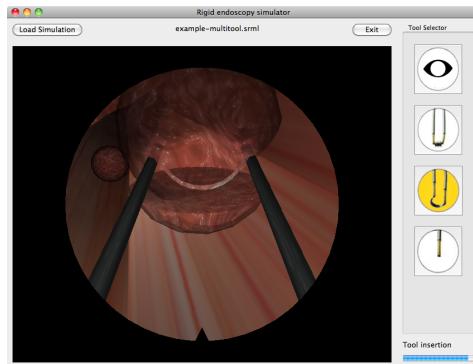
8. Pruebe ahora a utilizar la aguja —la herramienta de abajo. Antes de cambiar de herramienta, presione el pedal izquierdo para extraer la herramienta actual hasta que la barra de inserción esté completamente vacía. Esta herramienta tiene la característica de que puede ser accionada con las pinzas del dispositivo. Pruebe a realizar una biopsia del pólipos que hay a la derecha.

Controles alternativos:

- A** Accionar la herramienta.



9. Por último, la cuchilla —segunda herramienta— le permitirá cortar algún pólipos para su extracción. Puede probar a seccionar el pólipos anteriormente biopsiado.



10. Finalmente, para salir de la aplicación tan sólo tiene que hacer click en el botón *Exit* o cerrar la ventana.

7.3. Dudas frecuentes y resolución de problemas

En este apartado se da solución a ciertos inconvenientes que pudieran tener lugar con cierta facilidad durante el uso de la aplicación. Lea atentamente las instrucciones que debe seguir si se encuentra con alguno de estos problemas:

- **La aplicación va muy lenta.**

Además del renderizado gráfico, HysTrainer utiliza diversos algoritmos complejos para el cálculo de colisiones. Es por ello que, si la máquina en la que se ejecuta dispone de pocos recursos computacionales, estos cálculos tarden más de lo normal y su ejecución probablemente no pueda estar a la altura de lo esperado. Pruebe a ejecutar la aplicación en una máquina más potente.

- **No puedo cargar un fichero SRML. Me da un mensaje de error.**

Si ocurre esto es probable que esté intentando cargar un SRML distinto a los proporcionados en la instalación a modo de ejemplo. Tenga en cuenta que el formato SRML utilizado en HysTrainer es un formato propio de la plataforma ESQUI y es completamente independiente al estándar SRML de W3C (ver sección 2.3.3). En cualquier caso, estos son los errores que puede comunicar la aplicación y sus significados:

- *Bad scene configuration.* El fichero SRML no tiene la estructura correcta. Puede que falten elementos o que existan algunos que no debieran estar. Revise el estándar SRML que puede encontrar en el repositorio de datos de vtkESQui[8] o tome referencia de los ejemplos proporcionados junto a la aplicación.
- *Unknow object found.* La aplicación se ha encontrado en el SRML un objeto desconocido que no está clasificado como órgano ni como herramienta. Revise el contenido del fichero.

- *Invalid number of camera tools.* Cada documento SRML que describa un escenario de histeroscopia debe contar con un elemento que represente el volumen de la cámara. Este elemento se utiliza para el cálculo de colisiones de la lente. Si usted obtiene este mensaje, dicho elemento no se encuentra en el documento o existe más de uno.
 - *Missing texture in at least one element.* Todos los elementos del escenario deben tener una textura asociada. Se realiza así por sistema para evitar errores en tiempo de ejecución. Revise los modelos de visualización del documento en búsqueda de alguno que no tenga una textura asociada.
- **Cuando intento cambiar de herramienta, me aparece una ventana con el mensaje *You must extract the tool first*.**
Para poder cambiar de herramienta es necesario que la herramienta seleccionada actualmente esté totalmente extraída. Asegúrese de que el indicador de inserción (la barra de abajo a la derecha) se encuentra totalmente vacío antes de realizar el cambio de herramienta.

Capítulo 8

Manual del desarrollador

Este manual está orientado al desarrollador que desee trabajar sobre el software de este proyecto. Se tratará la instalación de la librería vtkESQui, así como la configuración del entorno y su uso básico. Se proporcionan ejemplos de código relativo al prototipo, incluyendo el uso junto a las librerías gráficas wxWidgets.

8.1. Instalación de vtkESQui

A continuación se describirá el proceso de instalación de la librería vtkESQui. Antes de comenzar, es requisito que el siguiente software se encuentre instalado en el sistema:

- CMake. Versión superior a la 2.5.
- Doxygen. Necesario para generar la documentación.
- VTK. vtkESQui no es compilable con versiones de VTK superiores a la 5.10. Es importante que se compile en modo 32 bits y con soporte para scripting en Python. Se recomienda la versión 5.10.
- Python. Versión 2.7 de 32 bits. La versión de 64 bits es incompatible con las librerías de VTK, vtkESQui y wxPython.
- Compilador. GCC en sistemas UNIX y Visual Studio 2008 en sistemas Microsoft Windows.
- Dispositivo periférico (opcional). Los controladores han de estar instalados si se desea hacer uso de él.

Para el proceso de instalación se requiere la descarga de los siguientes paquetes y códigos fuente:

- vtkESQui. Versión 0.7.1.[8].

- wxPython. Versión igual o superior a la 2.8.12[19]. Si se desea desarrollar una interfaz gráfica similar a la del prototipo de aplicación del proyecto. En el proyecto se utilizó la versión 2.8.12 y se encontraron problemas al intentar migrar a la versión 3.0.

Una vez instaladas las herramientas y paquetes necesarios, siga los siguientes pasos para compilar la librería *vtkESQui*:

1. **Descargue el código fuente** y extraiga el contenido del paquete comprimido en un directorio al que llamaremos *vtkESQui-src*.
2. **Cree un directorio** (lo llamaremos *vtkESQui-bin*) y **configure ahí el proyecto**.

- **En Linux:** sitúese en el directorio *vtkESQui-bin* y ejecute el siguiente comando:

```
cmake <vtkESQui-src>
```

- **En Microsoft Windows:** ejecute CMake y seleccione el directorio *vtkESQui-src* como directorio fuente y el directorio *vtkESQui-bin* como directorio de compilación.

Importante: preste especial atención al campo **VTK_DIR**, donde debe figurar el directorio de compilación de VTK. En sistemas Mac OS X, el campo **CMAKE OSX ARCHITECTURES** debe estar definido con la arquitectura **i386**.

Si se desea hacer uso de un dispositivo periférico especial, active el campo correspondiente al dispositivo en cuestión. Por ejemplo, para utilizar el dispositivo *Simball 4D*, se activaría el campo **VTKESQUI_USE_SBM**.

Para poder utilizar Python como lenguaje de scripting, active el campo **VTKESQUI_WRAP_PYTHON**. El campo **BUILD_SHARED_LIBS** deberá estar también activado.

Además, con el objetivo de poder generar la documentación posteriormente, active el campo **BUILD_DOCUMENTATION**.

3. Compile el proyecto.

En sistemas UNIX, ejecutando el comando **make** en el directorio *vtkESQuiBin*.

En Microsoft Windows, generando el proyecto en *Visual Studio 2008* en modo *release*.

4. Configure las variables de entorno necesarias.

PATH. Incluir aquí la ruta a los directorios de ejecutables de Python. Por ejemplo, en Windows y Para Python 2.7, añadir C:\Python27.

PYTHONPATH. Aquí se incluyen los siguientes directorios, necesarios para el funcionamiento correcto del wrapping en Python. Algunos no serán estrictamente necesarios en todos los sistemas, pero se añaden para asegurar la compatibilidad:

- a) Directorios de ejecutables y de wrapping de VTK.

```
<VTK-bin>/bin  
<VTK-bin>/lib  
<VTK-src>/Wrapping/Python  
<VTK-src>/Wrapping/Python/vtk  
<VTK-installed>/lib/site-packages (en caso de ser un paquete  
precompilado)
```

- b) Directorios de ejecutables y wrapping de vtkESQui.

```
<vtkESQui-bin>/bin  
<vtkESQui-bin>/lib  
<vtkESQui-src>/Wrapping/Python  
<vtkESQui-src>/Wrapping/Python/vtkesqui
```

LD_LIBRARY_PATH. Esta variable es necesaria en la mayoría de sistemas UNIX para la localización de librerías dinámicas. Configúrela añadiendo los directorios susceptibles de contenerlas:

```
<VTK-bin>/bin  
<VTK-bin>/lib  
<vtkESQui-bin>/bin  
<vtkESQui-bin>/lib
```

Si el objetivo final es generar un ejecutable a partir de un código Python (utilizando py2exe, tal como se hizo en el proyecto), en la variable PATH también ha de figurar el directorio donde se encuentran las librerías de Visual Studio redistribuibles.

8.2. Documentación del código Python

El código Python se documenta siguiendo las recomendaciones de las guías de estilo de Python[23] (*Python Enhancement Proposals* o *PEPs*). La documentación se encuentra embebida en el código y puede ser accedida en tiempo de ejecución o mediante introspección en el código. Además, la guía recomienda la escritura de código autoexplicativo, de manera que el mismo código se convierta en parte de la documentación.

La documentación en Python se escribe mediante las ristras de documentación (*docstrings*). Éstas ristras se escriben al comienzo de las funciones, clases y paquetes con el objetivo de ofrecer un resumen breve y conciso del propósito del código en cuestión. En tiempo de ejecución, las ristras de documentación se incluyen en los objetos y pueden ser accedidas a través del atributo “`__doc__`”.

El código Python de la aplicación prototipo HysTrainer se encuentra escrito y comentado siguiendo las directrices previamente comentadas.

8.3. Ejemplos de código de la aplicación

A continuación veremos un par de ejemplos enfocados al uso de `vtkESQui` para la simulación de técnicas de endoscopia rígida monocanal. El fin de dichos ejemplos es el de aprender a utilizar las nuevas funcionalidades de `vtkESQui` —que se implementaron en este proyecto—, así como la comprensión del funcionamiento de las características esenciales de esta librería y la integración con la interfaz gráfica de `wxPython` de la que se sirve el prototypo HysTrainer. Todos los ejemplos se abordan desde la capa de acceso Python de las librerías.

8.3.1. Carga de un escenario y puesta en marcha de la simulación

En este ejemplo, cargaremos un escenario de histeroscopia desde un fichero SRML con ayuda de la clase `vtkSRMLReader`, configuraremos algunos parámetros y pondremos la simulación en marcha.

En las dos primeras variables almacenaremos la ruta del fichero SRML y la inclinación de la lente, respectivamente. Además, será necesario que el atributo *DataPath* del fichero SRML esté apuntando a la localización del directorio `vtkESQuiData` en el equipo, donde se almacenan las mallas y texturas de los elementos de la simulación.

```
1 SRML_file_name    = "example-multitool.srml"
2 lens_angle = 30
3
4 import vtk
```

```

5 | from vtkesqui import vtkSRMLReader
6 |
7 | # Lectura del objeto simulacion
8 | reader = vtkSRMLReader()
9 | reader.SetFileName(SRML_file_name)
10| simulation = reader.ConstructSimulation()
11|
12| # Configuracion especifica del estilo.
13| simulation.GetInteractorStyle().SetLensAngle(lens_angle)
14|
15| # En este ejemplo desactivamos las colisiones.
16| simulation.InteractionOn()
17| simulation.CollisionOff()
18|
19| # Ponemos en marcha la simulacion
20| simulation.Run()

```

Como resultado, al ejecutar este ejemplo, debería aparecer una ventana VTK con los elementos del escenario cargado. Además, como la clase *vtkSRMLReader* también reconoce el tipo de simulación que se describe, el estilo de interacción característico de las técnicas de endoscopia rígida monocanal estará también asignado, por lo que podrá interaccionar con la escena de la manera descrita en el manual de usuario (capítulo 7).

8.3.2. Cálculo de colisiones con *vtkBioEngInterface*

En este ejemplo, un poco más avanzado, utilizaremos la clase *vtkBioEngInterface* para obtener el número de colisiones de la escena. Crearemos una escena con dos esferas que moveremos con el teclado, mientras en la consola se irá mostrando cada cuarto de segundo el número de colisiones.

En primer lugar, daremos valor a la variable que apunta al directorio de datos de ESQUI e importaremos los paquetes necesarios.

```

1 data_path = "<vtkESQuiData>/"
2
3 import sys
4 import vtk
5 from vtkesqui import *

```

A continuación definiremos dos funciones. La primera se encargará de atender los eventos de un temporizador que crearemos para la comprobación de colisiones. La segunda atenderá los eventos de teclado para desplazar las esferas de la escena y alejarlas o acercarlas.

```

7 def TimerCallBack(obj, event):

```

```

8     bioeng.Update()
9     cols = bioeng.GetNumberOfCollisions()
10    sys.stdout.write("\rColisiones: %d" % cols)
11    sys.stdout.flush()
12
13 def Keypress(obj, event):
14     key = obj.GetKeySym()
15     if key == "q":
16         sys.stdout.write("\r--- Fin ---\n")
17         quit()
18     elif key == "Up":
19         ele0.Translate(-0.1, 0, 0)
20         ele1.Translate(0.1, 0, 0)
21     elif key == "Down":
22         ele0.Translate(0.1, 0, 0)
23         ele1.Translate(-0.1, 0, 0)
24     scenario.Update()
25     scenario.Render()

```

Ahora cargaremos los modelos de visualización y colisión, para componer a continuación los elementos que conformarán los órganos de la escena. Cada órgano tendrá un único elemento y cada elemento poseerá un modelo de visualización y otro de colisión.

```

27 # Modelos de visualizacion
28 vis0 = vtkVisualizationModel()
29 vis0.SetFileName(data_path+"Scenario/Organs/ball.vtp")
30 vis0.SetTextureFileName(data_path+"Scenario/Textures/
31             leftball.jpg")
32 vis1 = vtkVisualizationModel()
33 vis1.SetFileName(data_path+"Scenario/Organs/ball.vtp")
34 vis1.SetTextureFileName(data_path+"Scenario/Textures/
35             rightball.jpg")
36 # Modelos de colision
37 col0 = vtkCollisionModel()
38 col0.SetFileName(data_path+"Scenario/Organs/ball_col.vtp")
39
40 col1 = vtkCollisionModel()
41 col1.SetFileName(data_path+"Scenario/Organs/ball_col.vtp")
42
43 # Elementos
44 ele0 = vtkScenarioElement()
45 ele0.SetVisualizationModel(vis0)
46 ele0.SetCollisionModel(col0)
47 ele0.SetPosition(-2,0,0)
48 ele0.Update()
49
50 ele1 = vtkScenarioElement()

```

```

51 | ele1.SetVisualizationModel(vis1)
52 | ele1.SetCollisionModel(col1)
53 | ele1.SetPosition(2,0,0)
54 | ele1.Update()
55 |
56 | # Organos
57 | org0 = vtkOrgan()
58 | org0.AddElement(ele0)
59 |
60 | org1 = vtkOrgan()
61 | org1.AddElement(ele1)

```

Creamos el objeto *vtkBioEngInterface* y añadimos los modelos de colisión que queremos que compute. Luego creamos el escenario y añadimos los objetos de la escena. Inicializamos ambos objetos y, en el caso del escenario, renderizamos una primera imagen.

```

63 | # Detección de colisiones
64 | bioeng = vtkBioEngInterface()
65 | bioeng.AddModel(colo)
66 | bioeng.AddModel(col1)
67 | bioeng.Initialize()
68 |
69 | # Escenario
70 | scenario = vtkScenario()
71 | scenario.AddObject(org0)
72 | scenario.AddObject(org1)
73 | scenario.Update()
74 | scenario.Render()

```

Finalmente creamos el temporizador y asignamos las funciones que anteriormente creamos a los eventos correspondientes, para por último iniciar el interactor y que comience la interacción.

```

76 | # Interacción con teclado
77 | iren = scenario.GetRenderWindowInteractor()
78 | iren.SetInteractorStyle(None)
79 | iren.CreateRepeatingTimer(250)
80 | iren.AddObserver("KeyPressEvent", Keypress)
81 | iren.AddObserver("TimerEvent", TimerCallBack)
82 | iren.Start()

```

El escenario creará automáticamente la ventana. Se podrán visualizar los vértices del modelo de colisión de cada esfera, pues no alteramos sus propiedades de visualización durante la instanciación. Utilizando las teclas **Arriba** y **Abajo** del teclado podrá alejar o acercar las esferas y, cuando se tocen,

podrá ver cómo aumenta el número de colisiones detectadas que se muestra en consola.

8.3.3. Integración con wxPython

En este último ejemplo veremos cómo integrar vtkESQui con wxPython, de manera que podamos cargar un SRML y ejecutar la simulación a través de una ventana de wxPython. El procedimiento será similar al del ejemplo 8.3.1, pero habrá que tener en cuenta ciertos aspectos relativos a la interfaz wxPython.

En primera instancia, especificamos la ruta del SRML y la inclinación de la lente y cargamos los módulos necesarios. Aquí se puede observar que se carga la clase *wxVTKRenderWindowInteractor*. Esta clase, comentada con más detalle en el apartado 5.3.2, permitirá la inclusión de una ventana de VTK en la interfaz gráfica y traducirá los eventos que ésta reciba.

```
1 SRML_file_name = "example-multitool.srml"
2 lens_angle     = 30
3
4 import wx
5 import vtk
6 from vtk.wx.wxVTKRenderWindowInteractor import
    wxVTKRenderWindowInteractor
7 from vtkesqui import vtkSRMLReader
```

Como utilizaremos una interfaz gráfica con su propio bucle de control de eventos, no podremos emplear el método *Run* de la clase *vtkSimulation* para poner en marcha la simulación. Hemos pues de crear un temporizador de wxPython, con el fin de que no haya conflictos al respecto.

```
9 class SimulationTimer(wx.Timer):
10     def __init__(self, simulation):
11         wx.Timer.__init__(self)
12         self.simulation = simulation
13
14     def Notify(self):
15         self.simulation.Initialize()
16         self.simulation.Step()
17         self.simulation.Render()
```

A continuación definimos la ventana de wxPython que será la ventana principal de la aplicación. En el constructor crearemos una instancia de la clase *wxVTKRenderWindowInteractor*, luego realizaremos el mismo procedimiento de carga del ejemplo 8.3.1 y finalmente pondremos en marcha el temporizador.

```

19 class MyFrame(wx.Frame):
20     def __init__(self):
21         wx.Frame.__init__(self, None, -1, "wxPython-
22                         vtkESQui example", size=(800,600))
23
24         # Insercion del elemento VTK en la ventana
25         # wxPython
26         widget = wxVTKRenderWindowInteractor(self, -1)
27         sizer = wx.BoxSizer()
28         sizer.Add(widget, 1, wx.EXPAND)
29         self.SetSizer(sizer)
30
31         # Lectura del objeto simulacion
32         reader = vtkSRMLReader()
33         reader.SetFileName(SRML_file_name)
34         simulation = reader.ConstructSimulation()
35
36         # Configuracion especifica del estilo.
37         simulation.GetInteractorStyle().SetLensAngle(
38             lens_angle)
39
40         # En este ejemplo desactivamos las colisiones.
41         simulation.InteractionOn()
42         simulation.CollisionOff()
43
44         # Asociamos el renderizado a esta ventana
45         renWin = widget.GetRenderWindow()
46         renWin.AddRenderer(vtk.vtkRenderer())
47         simulation.GetScenario().SetRenderWindow(renWin)
48
49         # Ajuste, centrado y mostramos la ventana.
50         self.Layout()
51         self.Center()
52         self.Show()
53
54         # Temporizador de la simulacion
55         self.timer = SimulationTimer(simulation)
56         self.timer.Start(50, wx.TIMER_CONTINUOUS)

```

Por último, y una vez definidas las clases que utilizaremos, la aplicación se construye y se pone en marcha con las siguientes tres líneas:

```

55 # Ponemos en marcha la aplicacion
56 app = wx.PySimpleApp()
57 frame = MyFrame()
58 app.MainLoop()

```

Es probable que en este ejemplo ocurran errores graves en el momento de

utilizar el teclado. Esto viene derivado de lo comentado en el apartado 5.3.2. En este caso, no hemos corregido la funcionalidad para preservar la simplicidad del código, pero puede ver un ejemplo en el módulo *SimulationRenderWindowInteractor* del código Python de HysTrainer.

8.4. Desarrollo futuro de vtkESQui

Durante la implementación del proyecto, hubo una serie de aspectos que no pudieron depurarse o implementarse, bien por cuestiones de tiempo, por complejidad o porque simplemente no entraban en el ámbito del proyecto. Tales aspectos se detallan en este apartado.

Además, se comentarán las vías futuras de desarrollo que se observaron gracias a la introspección realizada durante el proceso de desarrollo del prototipo de aplicación del proyecto.

En esta sección se asumirá que la versión actual de vtkESQui es ya la última versión (v0.7.1), producto del trabajo realizado en este proyecto y, por lo tanto, con las características previamente comentadas formando parte de la librería (ver apartado 5.2).

8.4.1. Directorio de datos

Actualmente, el programador debe especificar explícitamente la localización del directorio de datos de ESQUI (*vtkESQuiData*). Sería deseable que, en tiempo de compilación, se determinara el valor de una variable `VTKESQUIDATAPATH` que apuntara al directorio de datos de vtkESQui y pudiera ser referenciada tanto desde el nivel de código C/C++ como desde el nivel de scripting. Esto facilitaría la localización de las mallas y texturas para la construcción de los escenarios y podría simplificar las operaciones relativas.

8.4.2. Estilos de interacción

Control centralizado de eventos

Actualmente, los estilos de interacción están diseñados para permitir la interacción a través de ratón y teclado. Sin embargo, esto se utiliza realmente con fines de depuración, mientras que el objetivo final es que el medio de interacción principal del usuario con la simulación sea el dispositivo periférico. Siendo esto así y con la configuración actual, a menudo aparecen problemas de coherencia de datos y de sincronía, debido en mayor medida a la duplicidad de información que se almacena y modifica tanto en el estilo de interacción como en el objeto del dispositivo periférico.

Esto motiva que se proponga el estilo de interacción como objeto mediador de todos los eventos de interacción en la simulación y como único actor

sobre la escena, manteniéndose un control centralizado de los eventos y de las acciones que se llevan a cabo en la simulación. Esto evitaría tanto los problemas de incoherencia de datos como de sincronía, ya que las operaciones serían atómicas al utilizarse un solo hilo de ejecución.

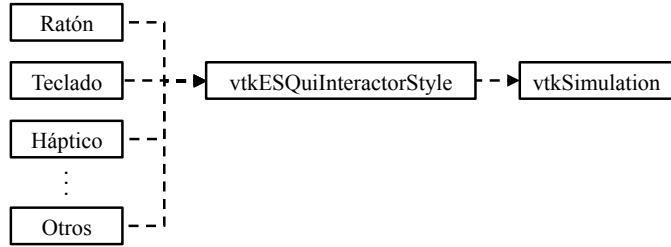


Figura 8.1: Esquema mostrando el estilo de interacción haciendo de interfaz de eventos.

El estilo de interacción deberá disponer pues de una interfaz a través de la que se actuará sobre la simulación. De esta manera, por ejemplo, el dispositivo periférico no actuará directamente sobre las herramientas de la escena, sino que transmitirá los parámetros de movimiento a través de la interfaz del estilo de interacción y éste será el que implemente las transformaciones correspondientes.

La implementación de estos aspectos resultará en un aumento de la cohesión de las clases de vtkESQui involucradas en la interacción y en una disminución drástica del acoplamiento de las mismas.

Restructuración de la jerarquía de clases de los estilos

Los únicos dos estilos de interacción de vtkESQui heredan de la clase *vtkInteractorStyleTrackballCamera*, pues provee de cierta base para la implementación de los mismos. No obstante, el hecho de que cada estilo herede directamente de esta clase propicia que la interfaz entre estilos no tenga por qué acogerse a ningún tipo de estandarización, pudiendo albergar grandes diferencias entre ellas (no ha de olvidarse que vtkESQui es un proyecto de software libre y, por lo tanto, distintos desarrolladores podrían implementar las mismas acciones de distinta forma).

En base a estas razones, es deseable que se implemente una clase abstracta que actúe de base e interfaz homogénea para los estilos de interacción. Así no sólo se procura la uniformidad de estilo en el código y de funcionamiento, sino que además se disminuye la duplicidad de código y se establece una clase que identificará a todos los estilos interacción de vtkESQui.

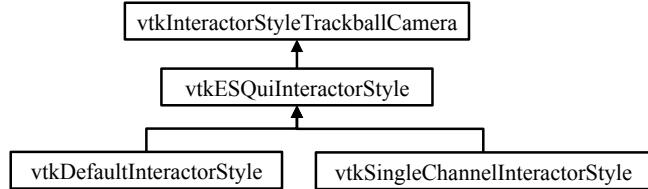


Figura 8.2: Diagrama de herencia de la propuesta.

Esto último es más importante de lo que pudiera parecer, pues como se observó en el apartado 5.2.1 es deseable no restringir demasiado el estilo de interacción para con la simulación, pero sí que es beneficioso a efectos de control y de prevención de errores garantizar que sólo se pueden asignar los estilos de interacción propios de `vtkESQui`.

8.4.3. Dispositivos periféricos

Lo propuesto en este apartado va en relación a la cohesión y el acoplamiento de las clases, tal como se comentó en la sección 8.4.2. Actualmente, `vtkESQui` da soporte a dispositivos periféricos de distintos fabricantes, pero cada uno de estos dispositivos tiene su propia clase e interfaz, con poco nivel de abstracción con respecto a la API que proporciona el fabricante.

Sí existe una clase base para todos ellos (`vtkHaptic`), pero en estos momentos se encuentra infrautilizada y su utilidad se limita a especificar la interfaz de un conjunto reducido de operaciones. Luego, cada clase derivada tiene sus propios métodos para acceder a las coordenadas del trócar, dándose incluso el caso de que una clase proporcione funcionalidades que el resto no hace.

Para esta línea futura se propone crear una especificación completa de interfaz en la clase `vtkHaptic`, de manera que se pueda realizar una abstracción efectiva del tipo de dispositivo periférico utilizado. Dicha interfaz estandarizada debiera contar al menos, y aparte de los que proporciona actualmente, con los siguientes métodos:

- Refresco de las coordenadas del dispositivo.
- Obtención de las coordenadas del trócar.
- Obtención de información acerca de la herramienta. Aquí sería deseable utilizar los tipos enumerados que se definen en las clases `vtkTool` y derivadas para designar los distintos tipos de herramienta de una manera única e independiente del dispositivo.
- Asignación de las fuerzas de resistencia del dispositivo, que serían de utilidad si éste fuera háptico.

Como puede observarse, y partiendo de lo tratado en el apartado 8.4.2, no se especifica un método que actualice directamente el escenario como ocurre actualmente con el método *Update*. Para esta acción, no obstante, se plantean dos posibles aproximaciones que deberían estudiarse cuidadosamente antes de su implementación:

1. La clase del dispositivo periférico se comunicará con el estilo de interacción a través de la interfaz estándar diseñada a tal efecto para que éste actúe de la manera oportuna sobre el escenario de la simulación.
2. El estilo de interacción contará con un método *Interact* que solicitará información al dispositivo periférico a través de su interfaz estándar y, en base a esta información, efectuaría las operaciones pertinentes sobre el escenario de la simulación.

En cualquier caso, el objetivo de esta línea futura es aumentar la cohesión y disminuir el acoplamiento de las clases, simplificando el flujo de información y facilitando con ello los futuros trabajos de mejora sobre la librería.

8.4.4. Mejora del soporte al formato SRML

Actualmente, la parte de ESQUI relativa al SRML necesita una cantidad considerable de trabajo en lo que a la especificación y el tratamiento se refiere.

Desde el surgimiento de ESQUI, el formato SRML ha ido cambiando progresivamente sin contar con una especificación estable. Esto ha provocado que el tratado de los archivos con este formato sea un poco confuso y que a menudo exista en el documento información irrelevante o redundante. En el directorio de datos de ESQUI existe un documento de especificación DTD, pero éste está obsoleto y, de cualquier forma, en la versión actual no se realiza comprobación en base a este documento.

Por consiguiente, se proponen los siguientes aspectos a desarrollar en esta vía:

- Revisar la especificación SRML de ESQUI. Crear una especificación estable y extensible, sin información redundante ni irrelevante. Poner especial énfasis en la legibilidad del documento. Si es necesario, desechar la especificación actual y partir de cero.
- Detallar la especificación, si es posible, en un documento DTD y que éste sea empleado por la clase *vtkSRMLReader* para la verificación de la estructura de los ficheros SRML.
- Dar soporte para la escritura de ficheros SRML a partir de un objeto de simulación. Creación de la clase *vtkSRMLWriter*.

8.4.5. Ampliación y mejora del sistema de colisiones y periférico

La última versión estable de vtkESQui proporciona la siguiente información en relación a una colisión entre dos elementos del escenario:

- Tipo de colisión. Se distingue entre colisión entre órgano y herramienta y colisión entre herramientas.
- Identificador del objeto involucrado.
- Identificador del modelo de colisión involucrado.
- Identificador de la celda involucrada.
- Normal de la celda involucrada.
- Identificador del punto de colisión en el modelo de colisión.
- Identificador del punto de colisión en el modelo de deformación.
- Normal del punto de colisión.
- Vector de dirección de desplazamiento de la colisión.
- Distancia de desplazamiento de la colisión.

En conjunto, esta información debería ser suficiente para reproducir deformaciones fidedignas a las colisiones visualizadas y detectadas. Sin embargo, el vector de dirección de desplazamiento de la colisión se calcula en base a la normal de la celda involucrada y no en base a la colisión en sí. Esto es motivo de que, actualmente, las deformaciones se produzcan en un sólo sentido independientemente de la dirección de la colisión. Además, este comportamiento puede ocasionar deformaciones contraintuitivas si se da el caso de que un elemento no es convexo.

Dadas estas circunstancias, se propone una revisión del cálculo de colisiones con el objetivo de implementar deformaciones realistas y consecuentes a las mismas. Una vez depurado este aspecto, dicha información podría emplearse para dar soporte al feedback háptico, el cual todavía no existe funcionalmente en vtkESQui.

Eventos de colisión

Por otro lado, también se observó la dificultad para implementar ciertos eventos que reaccionaran ante colisiones, exigiendo aproximaciones poco eficientes para tal propósito (inspeccionar las colisiones detectadas en busca de un objeto en concreto, por ejemplo).

Esto podría solucionarse mediante la implementación de un sistema de eventos para colisiones, de manera que pudieran asignarse observadores tanto a elementos concretos, como a objetos o a el escenario en su conjunto, y que éstos se activaran ante la colisión deseada.

8.4.6. Mejoras en el sistema de deformaciones y sincronización de mallas

Los sistemas que se emplean ahora mismo para la simulación de deformaciones son más que suficientes para casos en los que las deformaciones sean pequeñas con respecto al volumen del elemento. En este proyecto se realizaron pruebas con el sistema de masa muelle[21] (*vtkParticleSpringSystemInterface*) y se detectó que éste no es apto para cierto tipo de simulaciones porque no conserva el volumen de la malla deformada.

Por otro lado, se detectó que, en algunos casos, los elementos deformados no recuperan su forma. Se desconoce si la causa se localiza en el sistema de masa-muelle o en el de sincronización de mallas, pero la ocurrencia parece estar restringida al caso en que dos de los modelos del elemento tienen asignadas mallas idénticas.

Todo esto motiva un proceso de depuración del sistema actual de deformaciones y sincronización, así como el estudio de nuevos métodos que mejoren las cualidades de los presentes.

Capítulo 9

Conclusiones y líneas futuras del proyecto

En este apartado se exponen las conclusiones obtenidas a partir del trabajo en el proyecto y se discuten las posibles líneas futuras relativas al mismo.

9.1. Conclusiones

El prototipo de simulador de técnicas de histeroscopia *HysTrainer* resulta satisfactorio conforme a los términos que se establecieron en los objetivos y posteriormente en el estudio previo del mismo.

Esta aplicación prototipo, multiplataforma y de interfaz y uso sencillo, permite la carga de distintos escenarios a partir de documentos en formato SRML. En tales escenarios, el usuario puede comenzar a familiarizarse con las técnicas de endoscopia rígida monocanal, obteniendo una visión por pantalla similar a la real, e interactuando preferiblemente con la ayuda de un dispositivo periférico de interfaz humana orientado a la simulación virtual de técnicas sanitarias. Este dispositivo no ofrece feedback háptico, por lo que se resaltan visualmente las colisiones entre herramienta y órganos.

Durante la interacción, el usuario puede emplear las herramientas especificadas en el documento SRML, incluyendo cortes con cuchilla y herramientas activables mediante la acción de la pinza. Para todos estos procesos, la aplicación se sirve de las librerías VTK[2] y vtkESQui[1]. En concreto, esta última fue mejorada para dar soporte a las técnicas de endoscopia rígida.

La instalación en sistemas Microsoft Windows es muy sencilla y totalmente guiada. Para el resto de plataformas a las que se dio soporte (Mac OS X y Linux) la aplicación es autocontenido y no requiere instalación ninguna. En cualquier caso no existen dependencias de ningún tipo y la aplicación es utilizable directamente tras su instalación o copia en el equipo.

9.2. Líneas futuras

Este prototipo de simulador de técnicas de histeroscopia cuenta con la funcionalidad básica de un software de este tipo, si bien cuenta con ciertos aspectos a perfeccionar de cara a implantar su uso docente. A continuación se comentan dichos aspectos.

9.2.1. Líneas conceptuales

El prototipo de aplicación actual, si bien emula correctamente las características básicas de la técnica de histeroscopia, necesita de una labor adicional para adaptar su uso a la formación médica.

Tal proceso se basaría en el trabajo conjunto con profesionales médicos de la especialidad de ginecología y consistiría en la definición de ejercicios didácticos, así como de métricas que posibiliten la contabilización y puntuación de los ejercicios realizados. Para ello, también sería necesario el diseño de modelos 3D específicos para la recreación de escenarios característicos de la rama ginecológica, así como el instrumental propio de la especialidad.

Todo esto sería valorado en conjunto por los especialistas en la materia. Mediante un proceso continuo de retroalimentación, estos detalles de la aplicación se irían perfilando hasta conseguir una aplicación docente totalmente funcional. Tal aplicación podría ser utilizada para el entrenamiento específico de técnicas de histeroscopia y sería un recurso complementario a la formación médica clásica.

La aplicación final debería contar con una serie de ejercicios didácticos orientados a dominar las habilidades propias de la técnica en cuestión. En cada ejercicio se valorarían un conjunto de métricas, entre las cuales podrían estar contempladas, por ejemplo, las colisiones con las paredes uterinas, la consecución de ciertos objetivos y el tiempo empleado para ello. Podrían además presentarse distintos niveles de exigencia para cada ejercicio, de manera que el aprendizaje fuera lo más progresivo posible, enfocándose primeramente en los aspectos esenciales para posteriormente perfeccionar los detalles de la técnica.

Por otro lado, todo este proceso puede ser adaptable a otras especialidades médicas donde se empleen técnicas de endoscopia rígida monocanal, tomando como base el mismo prototipo producto de este proyecto.

9.2.2. Líneas de vtkESQui

En los aspectos técnicos, se ha podido comprobar que el software es completamente capaz. Sin embargo, y a tenor de lo que se trata en el apartado 8.4, se observaron ciertas vías de desarrollo en el código de vtkESQui que podrían

llevarse a cabo. Tales vías se comentan extensamente en el manual del desarrollador de este proyecto, pero podrían resumirse en los puntos siguientes:

- Creación de una variable que apunte al directorio de datos de `vtkESQui` (ver sección 8.4.1).
- Jerarquizar los estilos de interacción de `vtkESQui` y centralizar en ellos el control de eventos (ver sección 8.4.2).
- Estandarizar la interfaz de los dispositivos periféricos a partir de la clase `vtkHaptic` y hacer transparente y más sencillo el uso de distintos dispositivos periféricos (ver sección 8.4.3).
- Revisión del formato SRML para obtener una especificación estable y extensible y dar soporte a la escritura con una clase `vtkSRMLWriter` (ver sección 8.4.4).
- Mejorar el sistema de colisiones para proporcionar más información y permitir que se activen eventos a partir de las mismas (ver sección 8.4.5).
- Depuración del sistema de deformaciones e implementación de nuevos sistemas de deformación que preserven el volumen y la topología de la malla (ver sección 8.4.6).

Con la implementación de estas características se conseguiría una versión de la librería con unos niveles de potencia, flexibilidad y usabilidad muy superiores a los actuales. Dichos niveles permitirían fácilmente extender su uso a multitud de técnicas y especialidades distintas, así como equiparar la calidad de las simulaciones con los sistemas comerciales de simulación de técnicas sanitarias.

Apéndice A

Glosario de términos

- **Dolly, Azimuth, Elevation.** Coordenadas radiales de uso típico en el ámbito de las cámaras. Corresponden a la distancia con respecto el centro, el radio horizontal y el radio vertical.
- **Háptico.** Del griego háptō: tocar, relativo al tacto. Un dispositivo háptico es aquel que proporciona feedback táctil ejerciendo resistencia y/o fuerzas sobre el usuario.
- **Roll, Pitch, Yaw.** Equivalentes en español a alabeo, cabeceo y guiñada. Palabras tomadas de la jerga aeronáutica que se refieren a una rotación con respecto a los ejes **z**, **x** e **y** respectivamente. http://es.wikipedia.org/wiki/Mandos_de_vuelo
- **Trócar.** Instrumento en forma de punzón con una punta afilada en un extremo triangular. Utilizados típicamente dentro de un tubo hueco (cánula o manguito) para crear una abertura en el cuerpo y proporcionar un puerto de acceso durante la cirugía.
- **Wrapping.** Anglicismo: embalaje, envoltura. Método por el que se crea una interfaz para que determinado código pueda ser llamado desde un lenguaje distinto al lenguaje en que fue escrito. Lo normal es emplearlo para poder ejecutar funciones escritas en lenguaje compilado desde algún tipo de lenguaje interpretado.
- **Z-buffer.** Mecanismo mediante el cual se tiene en cuenta el posicionamiento de los objetos 3D de cara a una representación 2D de los mismos. Se trata de una matriz que almacena el valor de profundidad de los píxeles y permite evitar que objetos más lejanos a la cámara se superpongan en la imagen a objetos más cercanos.

Referencias bibliográficas

- [1] M.A. Rodriguez-Florido, N. Sánchez Escobar, R. Santana and J. Ruiz-Alzola. An Open Source Framework for Surgical Simulation. *Insight Journal*, 2006. <http://hdl.handle.net/1926/219>.
- [2] Visual toolkit. <http://vtk.org/>, Mar 2014.
- [3] Simball 4D. <http://g-coder.com/simball-4d>, Ene 2014.
- [4] wxWidgets. <http://wxwidgets.org/>, Nov 2013.
- [5] Python. <https://www.python.org/>, Mar 2014.
- [6] Blender. <http://www.blender.org/>, Mar 2014.
- [7] GNU Lesser General Public License. <http://www.gnu.org/licenses/lgpl-3.0.en.html>, Mar 2014.
- [8] Repositorio GIT de Motiva. <https://github.com/Motiva/vtkESQui>, Mar 2014.
- [9] Estándar SRML de W3C. <http://www.w3.org/TR/SRML/>, Mar 2014.
- [10] Mentice. <http://www.mentice.com/about-us/>, Mar 2014.
- [11] Immertion Corporation. <http://www.immersion.com/>, Mar 2014.
- [12] Immertion Corporation. Quarterly Report. June 30, 2002. <http://ir.immersion.com/secfiling.cfm?filngID=891618-02-3895>, Mar 2014.
- [13] CMake. <http://cmake.org/>, Ene 2014.
- [14] Dimitri van Heesch. Doxygen. <http://www.doxygen.org/>, Nov 2013.
- [15] Python Software Foundation License. <http://docs.python.org/2/license.html>, Mar 2014.
- [16] GIT. <http://www.git-scm.com/about>, Feb 2014.
- [17] Repositorio GIT del prototipo HysTrainer. <https://github.com/dcasbol/HysTrainer>, Mar 2014.

- [18] VTK File Formats. http://www.cacr.caltech.edu/~slombey/asci/vtk/vtk_formats.simple.html, Mar 2014.
- [19] wxPython. <http://wxpython.org/what.php>, Ene 2014.
- [20] I. Sommerville. *Ingeniería de software*. Pearson, 2012, ISBN: 6073206038.
- [21] Sistema de masa-muelle. [http://es.wikipedia.org/wiki/Masa_efectiva_\(sistema_masa-muelle\)](http://es.wikipedia.org/wiki/Masa_efectiva_(sistema_masa-muelle)), Abr 2014.
- [22] VTK/Python Wrapper Enhancement. http://www.vtk.org/Wiki/VTK/Python_Wrapper_Enhancement, Mar 2014.
- [23] Python Enhacement Proposal 257 – Docstring Conventions. <http://legacy.python.org/dev/peps/pep-0257/>, Mar 2014.
- [24] Xenophon Papademetris. An introduction to programming for medical image analysis with the visualization toolkit, 2006. http://medicine.yale.edu/bioimaging/suite/vtkbook/501_95557_xpvtkbook.pdf.
- [25] Kitware Inc. *VTK User's Guide*. Kitware Inc., 2010 ISBN: 1930934238.
- [26] Myriam García Berro y Concha Toribio. Fundación OPTI y FENIN. *Ciencias de la Salud. El Futuro de la Cirugía Mínimamente Invasiva. Tendencias tecnológicas a medio y largo plazo*. Cyan, Proyectos y Producciones Editoriales, S.A., 2004. http://www.fenin.es/pdf/prospectiva_cmi.pdf.
- [27] Cirugía mínimamente invasiva. http://es.wikipedia.org/wiki/Cirugía_mínimamente_invasiva#Siglo_XXI:_cirugía_laparoscópica, Dic 2013.
- [28] Historia de los endoscopios. http://www.olympuslatinoamerica.com/spanish/ola_aboutolympus_endo_esp.asp, Feb 2014.
- [29] Endoscopía. <http://es.wikipedia.org/wiki/Endoscopía>, Dic 2013.
- [30] Histeroscopia. <http://es.wikipedia.org/wiki/Histeroscopia>, Dic 2013.
- [31] Surgery simulator. http://en.wikipedia.org/wiki/Surgery_simulator, Dic 2013.
- [32] Kitware Inc. *Mastering CMake*. Kitware Inc., 2013, ISBN: 1930934262.
- [33] Mark Pilgrim. Dive into python, 2004. <http://www.diveintopython.net/>.
- [34] Encapsulador python para Microsoft Windows. <http://www.py2exe.org/>, Feb 2014.
- [35] Generador de instalador para Microsoft Windows. http://nsis.sourceforge.net/Main_Page, Feb 2014.

- [36] Encapsulador Python para Mac OS X. <http://pythonhosted.org/py2app/>, Feb 2014.
- [37] Encapsulador python para Linux. <http://www.pyinstaller.org/>, Feb 2014.
- [38] Guía del sistema LaTeX de composición de textos. <http://en.wikibooks.org/wiki/LaTeX>, Feb 2014.