Universitat Politècnica de Catalunya

Escola Tècnica Superior d'Enginyeria Industrial

GRAPHICS AND HAPTICS USER INTERFACE FOR VIRTUAL BRONCHOSCOPY

Director : Prof. Jan Rosell Gratacós

Author: Paolo Cabras

October 2010

Resumen

La broncoscopia virtual es una técnica de reconstrucción tridimensional generada por medio del ordenador que simula la vista de las vías aéreas , como si se estuviese llevando a cabo una broncoscopia real. Representa un ambiente útil de entrenamiento donde el doctor puede recibir indicaciones sobre el camino más corto que conecta la tráquea con la zona enferma que se necesita observar con más atención. La idea principal consiste en realizar una exploración de los pulmones lo mas real posible, como si el doctor estuviese sujetando un broncoscopio dando la sensación de estar interactuando con el ambiente real. Con este objetivo, el usuario llevará a cabo la tarea de exploración utilizando un dispositivo háptico, el cual permite la retro-alimentación de fuerzas que puede ser aprovechada para dar sensación de tacto y fricción y también para dar una posible guía a lo largo de una ruta calculada. El desarrollo de ese proyecto requiere: la implementación de una representación grafica de los pulmones desde un punto de vista exterior y desde la cámara del broncoscopio; la determinación de los movimientos del dispositivo háptico para poder controlar el broncoscopio virtual de la manera más real posible; la representación háptica de las fuerzas de contacto con los elementos virtuales y, finalmente, una evaluación por parte de un equipo médico especializado. El programa será desarrollado utilizando H3DAPI para la representación grafica y la interfaz háptica, Qt para construir la Interfaz Gráfica de Usuario y CMake para definir el proyecto: debido a esas elecciones la aplicación podrá funcionar en diferentes Sistemas Operativos.

Abstract

Virtual bronchoscopy is a computer-generated, three-dimensional reconstruction technique that simulates a bronchoscopist's view of the airways, creating a training environment where the doctor can be assisted with indication of the shortest path connecting the trachea to the lung's diseased zone which he wants to analyze.
The idea is to make the lung exploration as much real as possible, as if the doctor was really handling a bronchoscope, even giving the sensation of touching and interacting with the real physical bronchial tube. With this aim, in this work, the user will carry out the exploration task by using a haptic device, which is a device that allows force feedback that can be exploited to give touch and friction sensation and also to provide guidance along the calculated path.
The development of the project requires: a) The implementation of the graphical representation of the lungs both as a general view and as seen from the bronchoscope tip; b) The determination of the movement of the haptic device that allows to control the motions of the virtual bronchoscope in a realistic way; c) The haptic rendering of the contact forces; d) Validation with qualified personal. The software will be developed using H3DAPI for images representation and Haptic interface, Qt for the Graphic User Interface and CMake for defining the project: due to this choices the program will be able to run on different Operative Systems.

Contents

Preface

Lately, engineering and medicine are collaborating more and more to find new solutions to make the doctor task easier and increase the comfort for the patient. The tendency is to make the operations the least invasive possible and to intensify the use of virtual reality for training or diagnosis. This techniques have been improving their reliability and they are increasingly assuming a fundamental role on the medicine scene. Despite this big steps, many invasive medical procedure are still "unavoidable", an example is bronchoscopy. It still represents the unique way to obtain samples of the possible tumoral tissue for further analysis or to remove foreign objects from the airways. This procedure is still very bothering for the patient and in some cases it takes a long time (bronchoscopy can last between 20 and 30 minutes depending on the procedure to perform).
Also in this field the doctor can receive an important assistance from technology. The most spread and affirmed tools used are Virtual Bronchoscopy and training platform. They take care of two different but both important sides of the bronchoscopy field. Training platforms provide medical staff with a very "real" environment where they can approach to the bronchoscopy in a safe way and experiment some kind of complications that can come out during the real procedure.
Virtual Bronchoscopy helps doctors to make preliminary studies on the conformation of the patient airway and choose the better solution procedure. The increasing diffusion of this tool can be considered a natural expression of the current tendency of the *"personalization of the health care"* whose basic idea is to use the patient's internal structure to plan the better procedure. Furthermore, nowadays the clearness of the information is another important point and with 3D reconstruction provided by virtual bronchoscopy, patient and doctor can have almost the same level of understanding.

This work wants to place itself exactly in the junction of these two fields (training and virtual reconstruction to previous planning and diagnosis) creating a new tool that can "personalize" also the training.

Chapter 1

Introduction

Bronchoscopy is an interventional medical procedure used to analyze the bronchial tube, to see abnormalities of the airway, obtain samples from a specific lung site identified on preoperative images (CTI or MRI) [1], evaluate a person who has bleeding in the lungs, possible lung cancer, a chronic cough, or a collapsed lung, remove foreign objects lodged in the airway or open the spaces of a blocked airway.

A critical point for this technique is to find the right path to get to the target: doctors usually analyze the DICOM images [2] and try to plan the path on these 2-D images, which are very different from the real scene they will face while executing bronchoscopy. Finding the way inside the bronchial tree, then, comes to be very difficult and it takes a long time to the doctor to find the right path or to make an overall analysis of the airways, with a consequent great bother for the patient. Concerning this last point, the current tendency in medical procedures is to make them as much no-invasive as possible so that the patient would be less affected and would feel more comfortable. Considering this fundamental question, Virtual Bronchoscopy (VB) could be considered a useful tool. VB is a computer-generated, three-dimensional reconstruction technique that allows medical staff to explore the bronchial tree. This technique is already used to make a previous analysis of lungs, help individuating the diseased zone and to see if it is necessary to proceed with real bronchoscopy.

This project intends to improve Virtual Bronchoscopy so that to make the lung exploration as much real as possible, as if the doctor was really handling a bronchoscope, even giving the sensation of touching and interacting with the real physical bronchial tube. With this aim, in this work, the user will carry out the exploration task by using a haptic device, which is a device that allows force feedback that can be exploited to give touch and friction sensation.

[1] Computed Tomography and Magnetic Resonance Images
[2] Digital Imaging and COmmunications in Medicine, the set of slices coming from CT images (in gray scale)

Chapter 2

State of Art

2.1 Bronchoscopy

Bronchoscopy is a procedure in which a thin tube is threaded through the nose or mouth into the windpipe and lungs. This allows the clinician to look inside a patient's airway for abnormalities like blockages, bleeding, tumors, or inflammation. The clinician often takes samples from inside the lungs: biopsies, fluid, or endobronchial brushing. The clinician may use either a rigid or a flexible bronchoscope.

Types

There are two ways to perform a bronchoscopy, depending on which bronchoscope is used: a rigid bronchoscope or a flexible fiberoptic bronchoscope. We will center on flexible bronchoscope, since, today, the majority of bronchoscopies are performed using the flexible fiberoptic scope because of the improved patient comfort and reduced use of anesthesia. For the same reason, in our study, it has been decided to model the movements of this kind of bronchoscope.
Flexible fiberoptic bronchoscopy uses a fiberoptic camera on a flexible mounting to look inside the airway. This technique is more comfortable for the patient and does not require general anesthesia, although, in most cases, conscious sedation ("twilight sleep") is utilized. The bronchoscope can be considered as a device with 3 DOF, as it is shown in the picture 2.1:

- insertion/extraction of the tube (A).
- rotation along the axis of the tube (D).
- rotation ($\pm 90^o$) of the tip, that allows the doctor to turn. The rotation of the tip is made by rotating a little wheel (B) in the handle of the bronchoscope.

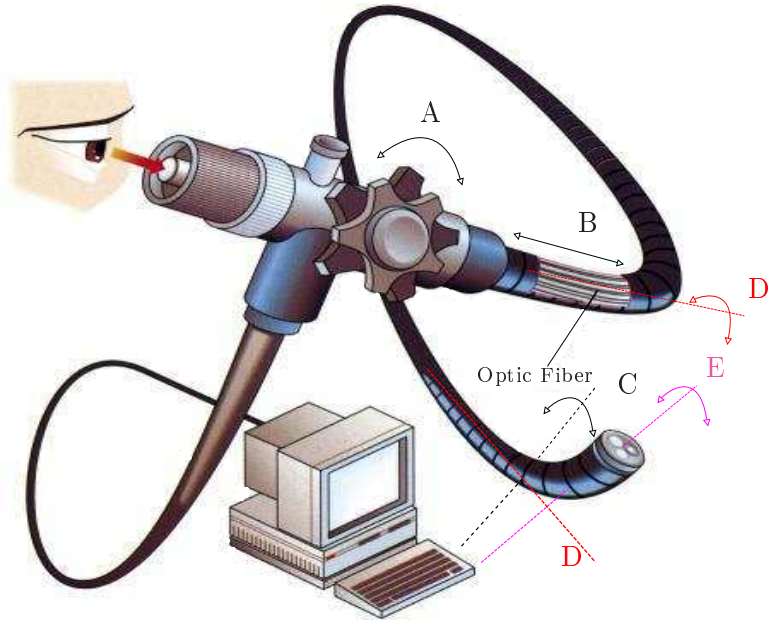


Figure 2.1: Degrees of freedom of a bronchoscope. 1) translational motion forward and backward along the D-axis generated by pushing and pulling the bronchoscope (B); 2) rotational motion around the C-axis obtained by turning wheel A (the camera central axis, E, coincides with D when the wheel is at its home position); 3) rotational motion around the D-axis produced by the rotation of the whole bronchoscope handle.

The bronchoscope is inserted through the nose (or mouth), down the throat, and into the airways (see fig. 2.2). The bronchoscope can also be inserted into the airway through a breathing tube. A flexible bronchoscope is a long, thin tube of optical fibers that transmit light. Before the procedure, the nose and throat are numbed by a spray of medicine squirted into those areas. This helps prevent coughing and gagging when the bronchoscope is inserted. The vocal cords, windpipe and airways are visualized by a light and mini-camera situated on the tip of the bronchoscope. Pre-medication is given before the procedure to relax the patient but not cause loss of consciousness.

As mentioned before, the navigation in the bronchial tree and, consequently, the diagnosis can be very difficult: because of their dimension and low flexibility, standard bronchoscopes cannot reach all the zone of the lungs (above all the upper and boarder zones), so the doctor is not able to see directly all the part of the lungs themselves. In the exploration with the bronchoscope, the doctor have not many references that helps him to find the right path towards the zone of interest, so the navigation is very difficult, sometimes. New advanced methods have been introduced to help navigate the bronchoscope and improve diagnostic ability, such as Virtual Bronchoscopy (which will be illustrated in the next section), *electromagnetic navigation* and *endobronchial ultrasound*. Electromagnetic navigation bronchoscopy detects lung tumors by using low-frequency electromagnetic waves to guide the bronchoscope.

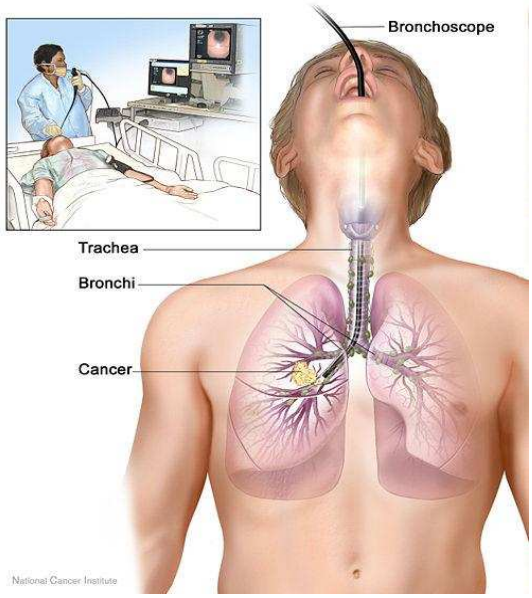


Figure 2.2: Scheme of flexible bronchoscopy

Endobronchial ultrasound can be combined with electromagnetic navigation bronchoscopy to achieve a great technique for tumor detection: a study found that the combination of the two techniques was able to successfully diagnose lesions in 88% of patients, whereas electromagnetic navigation bronchoscopy alone successfully diagnosed lesions in 59% of patients [1].

2.2 Virtual Bronchoscopy

Advances in computer technology have permitted development of virtual reality images of the tracheobronchial tree using data sets derived from helical CT of the chest. One method is *virtual bronchoscopy* (VB), which is a form of computed tomography (CT).

Virtual bronchoscopy is used for the screening of airways in endoluminal neoplasias diagnosis, for the evaluation of bronchial stenosis and to obtain a sort of map for the prospective fiberoptic bronchoscopy. This technique does not need the insertion of any instrument in the airways. It uses special x-ray equipment to take clear, detailed pictures of the inside of the lungs. Its 3D reconstruction allows a better observation of the relation between the airways and all the other organs or other structures. It has improved the ability of bronchoscopy to detect small, cancerous lung lesions that are inaccessible by conventional methods, above all in peripheral zones, but it cannot be used to sample neither cytologic nor histologic material. VB does not represent an alternative to fiber-optic bronchoscopy. It seems clear that the virtual bronchoscopic view of the airway, if both anatomically accurate and providing accurate measurements, is an important new addition in

the pulmonary physician's practice, and is especially useful in evaluating the airway before initial or sequential real bronchoscopies where an interventional procedure might be considered. These images reduce the chances of significant surprises occurring during any procedure, and allow the patient and family to be as fully informed of the records as the physician is.

Complex Virtual Bronchoscopy

Virtual bronchoscopy can be used to achieve different results and it is becoming the current helping tool to improve diagnosis and to facilitate the task of the bronchoscopy. We can basically distinguish three complex application of the VB idea.

The first of these complex virtual bronchoscopy applications uses the data contained within the three-dimensional image for procedures of *localization* within the mediastinum[1] and hilar structures[2].

A simple example is understanding where a mediastinal lymph node is in relation to the bronchial tree. In traditional bronchoscopy, the doctor can only see the airway lumen and not on the other side of the nontransparent bronchial epithelium, where the lymph node (that is the target for a transbronchial needle aspiration or core biopsy) may be. Using the virtual bronchoscopy information obtained from the three-dimensional MDCT (Multi Detector Computed Tomography) dataset, it can be shown precisely the same image as the real bronchoscope does, with gray-scale rendering. With these two images side by side (the virtual bronchoscopic visualization and the real bronchoscope visualization of the same region), it is possible to make the "virtual bronchial wall" transparent, so that the structures through the bronchial wall can be visualized.
Once the lymph node is made visible on the virtual bronchoscope images, comparing the virtual and real scene, the bronchoscopy operator can, with a great degree of confidence, know where the target lymph node is. His task can be made easier using a computer script for the image processing to transfer the virtual image with the lymph node in question to overlay the real bronchoscopic image.
Early results suggest a greater than 90% success rate for mediastinal and hilar lymph node biopsies. Clearly, these early successes open the possibility of sampling mediastinal structures precisely and

[1] The mediastinum is a non-delineated group of structures in the thorax, surrounded by loose connective tissue. It is the central compartment of the thoracic cavity. It contains the heart, the great vessels of the heart, esophagus, trachea, phrenic nerve, cardiac nerve, thoracic duct, thymus, and lymph nodes of the central chest.

[2] Above and behind the cardiac impression is a triangular depression named the hilum, where the structures which form the root of the lung enter and leave the viscus. These include the pulmonary artery, the superior and inferior pulmonary veins, lymphatic vessels and the bronchus, with bronchial vessels surrounding it.A pleural sleeve is created around these structures, where the pleura reflects, changing from visceral to parietal. These structures pass through the narrow hila on each side and then branch as they widen out into the lungs. The hila are not symmetrical but contain the same basic structures on each side.

reliably. This also opens up the possibility of localized application of nonspecific or specific therapies.

A second advanced virtual bronchoscopic application is that of *path-finding* to a peripheral region of interest within the lung [2]. With the significant improvement in MDCT scanners, seven or eight generations of airways can now be automatically extracted and evaluated. A specific software can interrogate the three-dimensional image dataset and provide a pathway that links the trachea with the lesion in the lungs peripheral region. This path, then, can be easily cannulated and once this tube is placed, several probes can be placed either to brush or biopsy or optically or by ultrasound sample the lesion of interest.
Using these sorts of approaches, in early studies, 80% of peripheral lung lesions can be easily and satisfactorily sampled. These pulmonary pathfinding applications are in clinical studies and are being developed by a number of companies for medical application.

The third advanced virtual bronchoscopy application involves the targeting of the peripheral lung for endobronchial valve-type procedures in the management of pulmonary emphysema, so-called endobronchial lung volume reduction surgery [3]. Here, the information that is required is the state of the lung parenchyma and the extent of emphysema in each segmental region together with the anatomic configuration and size of the subtending airway segments. In this case, it allows operating interventional pulmonologist to plan many things before the procedure, such as valve size, how many segments a device might require to be placed, whether the segment lengths are adequate for the valve placement.

In summary, it can be affirmed that VB is getting a more and more useful tool to be used with actual bronchoscopy. Application of advanced virtual bronchoscopy techniques is an obvious solution to shorten procedure times, to improve accuracy of device placement, to reduce medical error, and to educate the patient and families.

2.3 Simulation Training

One important tool for the training of the bronchoscopy operator is the bronchoscopy simulator (2.3). It's a virtual reality simulator with a model flexible bronchoscope that looks and feels like the real object. It interacts with a small bench top sensor which registers the scope's movements and translates this to a three-dimensional virtual airway displayed on a computer screen. The user can progress through virtual clinical scenarios receiving instant feedback from the simulator which measures participant performance. In certain models the reconstruction of medical environment is very realistic, it can comprise a replica of a human body and many complications that can happens during the real bronchoscope.

A bronchoscopy simulator allows the doctors to repeat the procedural experiences at their own

Figure 2.3: The VR bronchoscopy skill station includes a cart, computer, display, printer, proxy flexible bronchoscope, and keyboard. While the operator performs Flexible Fiber Bronchoscope via the left nares, realistic resistance is felt during manipulation of the proxy flexible bronchoscope.

pace and practice medical procedures by accurately duplicating the look and the feel of real-life situations. This improves patient safety by allowing the doctor to become better trained without putting patients at risk. The study in [4] compared the bronchoscopy skills and cognitive knowledge of 22 fellows who received standard bronchoscopy training with 22 fellows who received additional bronchoscopy training, including simulation bronchoscopy and an online curriculum. Results showed that fellows who received additional simulation training significantly improved their bronchoscopy skills and accelerated the acquisition of skills compared with those who received standard training.

Simulators will allow this initial training to occur in a time-efficient and cost-effective manner. In a very short period of time, fellows can be exposed to a broad range of cases that reflect variations in patient anatomy, pathology, and physiology. These exercises or procedures would otherwise require numerous real-life encounters and costly hours of supervision.

Simulation Procedure

To begin the bronchoscopy, the user inserts the bronchoscope into the robotic device. The bronchoscope feels and acts like an actual flexible fiberoptic bronchoscope. The device tracks the motions of the flexible bronchoscope and reproduces the forces felt during an actual bronchoscopic procedure. The proximal end of the interface device can be shaped like a human face with a port to insert the flexible bronchoscope through one of the nasal passages. The flexible bronchoscope tracks the manipulations of the tip control lever, the suction button, and video buttons. In addition, instruments are tracked as they are manipulated in the working channel. This allows for biopsies and other diagnostic and therapeutic procedures to be performed on the simulator.

A monitor displays computer-generated images of the airway as the user navigates through the virtual anatomy. Texture maps based on videotapes of actual bronchoscopic images are added to the airway models to give the mucosa a realistic look. Using different CT scan data sets allows for the development of a variety of simulated cases that reflect a range of patient anatomy and pathology.

In addition to being anatomically correct, the virtual patient also behaves in a realistic manner. The patient breathes, coughs, bleeds, and exhibits changes in vital signs. Complications are programmed in such as lidocaine toxicity causing the patient to seize or develop a cardiac arrhythmia.

The simulation computer software records all the actions of the user and stores this information in a database. Complications such as hemorrhage, pneumothorax and cardiorespiratory distress can be programmed to occur during a simulated case. The trainee must then respond in a timely and appropriate manner.

Chapter 3

Motivation

In comparison with real bronchoscopy, Virtual Bronchoscopy has some advantages. It is a non-invasive procedure that can visualize areas inaccessible to the flexible bronchoscope. Virtual bronchoscopy is able to evaluate bronchial stenosis and obstruction caused by both endoluminal pathology (tumor, mucus, foreign bodies) and external compression (anatomical structures, tumor, lymph nodes), can be helpful in the preoperative planning of stent placement and can be used to evaluate surgical sutures after lung transplantations, lobectomy or pneumectomy. Virtual bronchoscopy, then, is a very useful tool but it provides only an exploration made by a camera passing through points, which are established previously. Since all the commands are given by keyboard or mouse, this procedure does not allow any sort of interaction with the tissue nor a correspondence with the real movements the doctor would do with the bronchoscope. Furthermore, since no interaction is provided, the "virtual camera" can go out of the bronchial tree, while exploring within.

The bronchoscopy simulator, which, on one side, provides a very realistic environment (considering force feeling, procedure and instruments), is not very practical, on the other hand, for a specific analysis on a particular patient: it provides only a range of cases that reflect variations in patient anatomy, pathology, and physiology. Moreover, it does cost a lot.

Therefore, there is the need of a VB system able to be navigated in a more realistic way. This can be achieved exploiting the characteristic of haptics devices that can provide the doctor with a complete platform to see and, somehow, feel as if he were doing a real bronchoscopy on that specific patient.

The natural target of this software will be bronchoscopy experts, who will exploit the useful characteristics of the VB joined with a more ergonomic way of doing it.

By using VB with haptic feedback, medical staff is expected to carry out their tasks in a more fluent way and in a shorter time, since they would know the path and movement in advance. Fur-

thermore, this fact is also expected to improve the patient reaction to this operation, since it would take less time and reduce the number of "forward-backward" movements of the bronchoscope in bronchis, which are very annoying for the patient.

The present work is focused on the development of a system able to allow people to navigate inside the bronchial tree using an haptic device. Since it is part of a bigger project, this work also responds to the need of having a software that could integrate all the tools developed in the whole project: load the 3D-lung reconstruction from CT images and the shortest path calculated to get to the diseased lung zone and allow people to navigate the 3D scene using a haptic device.

This work intends to represent another little step to the "*personalization of health care*", which means to use the patient's internal structure and function for the training or preplanning of complex diagnostic or therapeutic procedures. This seems to be a fundamentally important characteristic for improving the outcome of health care.

Chapter 4

Objectives

The main objective of this work is to make a basic application which, exploiting virtual reality and haptics device potentialities, can be used to execute virtual bronchoscopy of that particular patient in a more practical and realistic way, using the same movements to move the camera within the airways and recreating, somehow, the feelings of a real bronchoscopy. It is not the aim of this work to build a complex simulator.

To achieve this objective, this work is divided into two modules:

- **Navigation Module**
  The first problem that has to be resolved is the determination of the haptics device movements that allow to control the motions of the virtual bronchoscope in the most realistic way, in other words the most accurate correspondence between camera motion controlled by haptics device and the camera motion controlled by broncoscope has to be found, disabling, if it is necessary, those device DOF not present in the real broncoscope.

  This module must also cope with the rendering of the contact forces and the management of the collisions, in order to avoid the exit from within the bronchial tube.

- **Visualization Module**
  The software also requires a Graphics Interface that permits the user to see the virtual scene and explore it: to make the exploration more intuitive, the application must provide the graphical representation of the lungs both as a general view and as seen from the bronchoscope tip, so the user can always know in which position he is and where to go.

Another important objective is to create a "cross-platform" software, able to run on different Operative System. Therefore, the libraries chosen for the implementation will meet this requirement.

Chapter 5

Involved Hardware: Haptic Device

5.1 Haptic Devices

Haptic devices (or haptical interfaces) are mechanical devices that allow users to touch, feel and manipulate 3D objects in a virtual environment or in a tele-operated system. Unlike the other most common computer interface devices (mouse, keyboard or joystick) which are input-only devices, the haptic devices are input-output devices, meaning that they track a user's physical manipulations (input) and provide realistic touch and sensations coordinated with on-screen events (output): they give the force feedback to the subject who is interacting with virtual or remote environments, giving, somehow, the sensation of being present in the scene.

The word haptic derives from the Greek *haptikos* meaning "being able to come into contact with". In human-computer interaction, haptic feedback means both *tactile* and *force* feedback. The term Tactile, or touch feedback is used to express the sensations felt by the skin. Tactile feedback allows users to feel things such as the texture of surfaces and vibration. Force feedback reproduces directional forces that can result from solid boundaries, the weight of grasped virtual objects, mechanical compliance of an object and inertia. All these features enhance the task performance and increase the realism of a simulation.

Considering the human sensorial characteristics, they impose much faster refresh rates for haptic feedback than for visual feedback: for example, tactile sensors in the skin respond best to vibrations higher than 300 Hz. For this reason HAPI[1], and usually also the other haptic rendering APIs manage the high priority threads for haptic rendering at 1000 Hz [5]. This order-of-magnitude difference between haptics and vision bandwidths requires that the haptic interface incorporate a dedicated controller. Since it is computationally expensive to convert encoder data to end effector position

[1] Library for haptic handling and rendering. It will be further described in section 6.

and translate motor torques into directional forces, a haptic device will usually have its own dedicated processor. This removes computation costs associated with haptics and the host computer can dedicate its processing power to application requirements, such as rendering high-level graphics [6].

The haptic interfaces sold at present can be classified as either ground-based devices (force reflecting joysticks and linkage-based devices) or body-based devices (gloves, suits, exoskeletal devices). The most popular design on the market is a linkage-based system, which consists of a robotic arm attached to a stylus. The arm tracks the position of the stylus and is capable of exerting a force on the tip of this stylus.

An alternative to a linkage-based device is one that is tension-based. Instead of applying force through links, cables are connected to the point of contact in order to exert a force. Encoders determine the length of each cable. From this information, the position of a "grip" can be determined. Motors are used to create tension in the cables, which results in an applied force at the grip. In this case, it has to be considered the one which better would represent the bronchoscope, so it had been chosen a linked based system.

Many kind of devices can be found among the linkage-based systems. Since the configuration of one bronchoscope is determined by the tip's position and orientation, we centered on the 6 DOF devices. Among the brands, we decided for PHANTOM by Sensable Technologies, whose devices were already used in our laboratory (PHANTOM Premium) and which was one of the first in commercing force feedback devices and, nowadays, is the most popular.

Among the PHANTOMs there are three devices that could suit our needs:

- Premium: 6 DOF and force feedback for the position and momentum for the orientation.
- Desktop: 6 DOF but force feedback for only 3 joints (the position ones), i.e. only three electric actuators to virtually fix a point in a 3D space. Also his workspace is smaller than the Premium's one.
- Omni: like the Desktop but with less power for the force-feedback and more friction.

Considering the degrees of freedom of the bronchoscope and above all the charateristic and wanting to make the doctor feel the bronchoscopy and, in the future, provide a sort of guidance, the 6 DOF of Phantom Premium would be the best choice. The momentum given by this device will increase the reality of the application and moreover it would allow to force the user to adopt a determined orientation, which will help a possible guidance to be more effective. Other factors, such as costs, has to be taken into account for the choice, because it would be kind of meaningless

Model	The PHANTOM Desktop Device	The PHANTOM Omni Device
Force feedback workspace	~ 6.4 W x 4.8 H x 4.8 D in > 160 W x 120 H x 120 D mm	~ 6.4 W x 4.8 H x 2.8 D in > 160 W x 120 H x 70 D mm
Footprint Physical area the base of device occupies on the desk	5 5/8 W x 7 1/4 D in ~ 143 W x 184 D mm	6 5/8 W x 8 D in ~ 168 W x 203 D mm
Weight (device only)	6 lb 5oz	3 lb 15 oz
Range of motion	Hand movement pivoting at wrist	Hand movement pivoting at wrist
Nominal position resolution	> 1100 dpi ~ 0.023 mm	> 450 dpi ~ 0.055 mm
Backdrive friction	< 0.23 oz (0.06 N)	< 1 oz (0.26 N)
Maximum exertable force at nominal (orthogonal arms) position	1.8 lbf. (7.9 N)	0.75 lbf. (3.3 N)
Continuous exertable force (24 hrs.)	0.4 lbf. (1.75 N)	> 0.2 lbf. (0.88 N)
Stiffness	X axis > 10.8 lb/in (1.86 N/mm) Y axis > 13.6 lb/in (2.35 N/mm) Z axis > 8.6 lb/in (1.48 N/mm)	X axis > 7.3 lb/in (1.26 N/mm) Y axis > 13.4 lb/in (2.31 N/mm) Z axis > 5.9 lb/in (1.02 N/mm)
Inertia (apparent mass at tip)	~ 0.101 lbm. (45 g)	~ 0.101 lbm. (45 g)
Force feedback	x, y, z	x, y, z
Position sensing [Stylus gimbal]	x, y, z (digital encoders) [Pitch, roll, yaw (± 3% linearity potentiometers)]	x, y, z (digital encoders) [Pitch, roll, yaw (± 5% linearity potentiometers)]
Interface	Parallel port and FireWire® option*	IEEE-1394 FireWire® port: 6-pin to 6-pin*
Supported platforms	Intel or AMD-based PCs	Intel or AMD-based PCs
OpenHaptics® Toolkit compatibility	Yes	Yes

Figure 5.1: Comparison table between Phantom Omni and Phantom Desktop

to build a program on a expensive device if an acceptable result can be achieved also with much cheaper devices (e.g. Phantom Omni that is 25 times cheaper than the Phantom Premium).

As it can be seen on the comparison table (fig. 5.1), Desktop and Omni basically differs only on the strength of the force feedback. Since it was seen that the force provided by the Omni was sufficient, finally PHANToM Omni was chosen for this application.

5.2 PHANTOM Omni

PHANTOM Omni has 6 DOF: 3 digital encoders to read the position and ±5% linearity potentiometers to read the stylus gimbal angles. As mentioned before, it cannot provide torque but only force feedback in the three orthogonal axis X, Y and Z.

Control Algorithm

The Omni (as the other PHANTOM devices) uses an impedance control algorithm: the user moves the device and it responds with a force, if it is necessary. In this kind of algorithm the control loop can be resumed as follow. The position sensors detect the movement made with the device by the user, then the hardware controller (driver) sends the translation information to the software simulator, which determines if a reaction force is required and its magnitude value. The host computer sends feedback forces to the device. Actuators (motors within the device) apply these forces based on mathematical models that simulate the desired sensations.
For example, when simulating the feel of a rigid wall, the driver orders to the motors to stay in a determined position and actuators (motors within the device) exerts the forces needed to stay at that position. The farther the wall is penetrated, the harder the motors push back the device.

Workspace Definition

Nominal Worspace (NW): it is the volume in front of the device, in which the manufacturer guarantees the specified force feedback and precision. No physics limitations forbid to exit this volume, since it is only definition of part of the space and the joint range allows wider movements. For the Omni device is a parallelepiped whose dimensions are $160\,W \times 120\,H \times 70\,D$ $[mm]$.

The nominal workspace is different from the **real workspace (RW)** which is the volume containing all the points reachable by the device end effector, exploiting the maximum range of all joints. The RW comprises marginal zones where the device behavior would not be acceptable for certain applications, this would be a question to take care of in every prospective work.

5.3 Haptic Renderers

The haptic renderer's task (or haptic-rendering algorithm) is to compute the correct interaction between the representation of the haptic interface in the virtual environment and the objects populating that envinronment. Furthermore, these algorithms ensure that the haptic device correctly renders such forces on the human operator.

Following the idea of professors Salisbury, Conti and Barbagli described in [7], three parts can be distinguished in a haptic-rendering algorithm:

- *Collision-detection* algorithms detect collisions between objects and avatars[2] in the virtual

[2]The virtual representation of the haptic interface through which the user physically interacts with the virtual environment.

environment and yield information about where, when, and ideally to what extent collisions (penetrations, indentations, contact area, and so on) have occurred.

- *Force-response* algorithms compute the interaction force between avatars and virtual objects when a collision is detected. This force approximates as closely as possible the contact forces that would normally arise during contact between real objects. Force-response algorithms typically operate on the avatars' positions, the positions of all objects in the virtual environment, and the collision state between avatars and virtual objects. Their return values are normally force and torque vectors that are applied at the device-body interface.
- *Control algorithms* command the haptic device in such a way that minimizes the error between ideal and applicable forces. These algorithms take as input the force and torque vectors computed by force-response algorithms and return the values of the actual force and torque vectors that will be commanded to the haptic device.

For example, HAPI (haptic rendering API which will be described better in section 6) is a library that allow to build haptic interfaces, giving methods and function which implement the three parts just mentioned. There are four haptic renderers available in this library: two are internally implemented in HAPI and two use external haptics libraries. Each of them can work better than the others on a specific shape model or in a specific application.

GodObject renderer

The *god-object* algorithm was introduced by Zillers and Salisbury [8]. It uses a point proxy and supports custom user defined surfaces. Moreover, it is Open Source and device independent but could have problems wth the concave part of a curved surface.

Ruspini renderer

The *RuspiniRenderer* is based on the algorithm presented by Ruspini et al. in [9]. It is different from all the other renderers in HAPI in that it uses a sphere proxy making it possible to have an interaction object with a size instead of just a point. As the GodObject, also this renderer is Open source, device independent and support user surfaces, but it is a little slower than the other renderers.

Chai3D

Chai3D is an open source haptics library distributed under the GNU GPL license. It has been developed by a team at Stanford University in California, USA. As the others, it is open source and

device independent, but this presents some fallthrough problems on moving objects and it does not allow user defined surfaces [10].

OpenHaptics

OpenHaptics is a proprietary haptics library developed by SensAble Technologies. It uses a point proxy based approach and provides a stable haptic feedback. It is however not very extendable in terms of user defined surfaces and only works with haptics from SensAble Technologies, moreover is closed source. Among the pros of this renderer, we can mention the availability of MagneticSurface[3] and a good behavior with moving objects.

[3] It is a H3DAPI surface node with a particular haptic property that makes the surface magnetic: the surface has a sort of magnetic field (modeled like a spring force, whose spring constant and the range of action can be set) that attracts the proxy and force it to stay on the surface, sending to the device the corresponding forces to make it stay on the surface. It works only with OpenHaptics Renderer

Chapter 6

Involved Software

6.1 CMake

As it is described in the official site [11], CMake is an extensible, open-source, cross-platform make system that manages the build process in an operating system and in a compiler-independent manner. CMake has many functionality: it can compile source code, generate wrappers, create libraries and build executables in arbitrary combinations. CMake also supports static and dynamic library builds.

Unlike many cross-platform systems, CMake is designed to be used in conjunction with the native build environment. Moreover, CMake generates a cache file that is designed to be used with a graphical editor. For example, when CMake runs, it locates include files, libraries, and executables, and may encounter optional build directives. This information is gathered into the cache, which may be changed by the user prior to the generation of the native build files.

It is very easy to use: the build process is controlled by creating one or more configuration files in each source directory (called CMakeLists.txt files) that make up the project. These configuration files are used to generate standard build files (e.g., makefiles on Unix and projects/workspaces in Windows MSVC) which are employed in the usual way. CMake provides many pre-defined commands but, if needed, user defined commands can be made. Furthermore, it is possible to add other makefile generator (Unix and MSVC++ is supported currently) for a particular compiler/OS configuration.

Its easy learning and all the possibilities it offers make CMake an almost essential (or at least very useful) tool for developing cross-platform software.

6.2 Graphic Rendering: H3DAPI

For graphic rendering there are many solutions too. Scene-graph based API seemed to be a good tool to make complex application more manageable, so we will center on this kind of API. The scene graph structure uses a hierarchical representation of a scene with nodes to divide the world into smaller sub-components with its characteristics and properties set in specific fields.

OpenInventor is an object-oriented 3D toolkit offering a comprehensive solution to interactive graphics programming problems. It can be defined the *de facto* standard for 3D visualization and visual simulation software in the engineering community. It presents a programming model based on a 3D scene database, including a rich set of objects (such as cubes, polygons, text, materials, cameras, lights, trackballs, handle boxes, 3D viewers) and editors that speed up the programming time. This toolkit is built on top of OpenGL, provides a stardard file format for data interchange and it is cross-platform, window-system and platform independent. It allows animations and creating new users customized objects.

Coin3D is another example of high-level 3D graphics toolkit for developing cross-platform real-time 3D visualization and visual simulation software. As OpenInventor is, also Coin3D is built on OpenGL and uses scene graph data structures to render 3D graphics in real-time. In fact, Coin3D is fully compatible with OpenInventor. In addition, it provides 3D sound, 3D textures, and parallel rendering on multiple processors. Another interesting feature is the seamlessly integration in the Qt development environment. Coin3D, which is only oriented to the graphic rendering, is a "very much developed" instrument to manage graphic rendering: it offers a wide range of tools to managing the camera and creating several indipendent window with different point of view perfectly integrated with Qt (thanks to the SoQt library[1]).

Both of these mentioned API only provide graphic-rendering, so in applications that require haptic devices, haptic rendering has to be developed in parallel. If you want to use scene-graph APIs, the solution is to create a duplication of the scene-graph, one cointaining the haptic specifications and the other for the graphic rendering. This would imply to keep the two in sync (Coin3D and OpenInventor have callbacks mechanism to handle this, unless you have many dynamic objects moving around) and may imply data redundancy (every object has to be duplicated in both scene-graph).

H3DAPI is an open-source, cross-platform, scene-graph API. H3D is written entirely in C++ and uses OpenGL for graphics rendering and HAPI for haptics rendering. It provides a scene-graph API that merge graphics and haptic features: it performs graphic and haptic rendering from a single scene description. H3D leverages the *de facto* industry standard haptic library OpenHaptics. Through the use of HAPI there is also haptics rendering support for several other devices that does

[1] For more information on this library, see http://doc.coin3d.org/SoQt/

not depend on OpenHaptics[2] . H3D is built using many industry **standards** [12] including:

X3D[3].The Extensible 3D file format that is the successor to the VRML standard. X3D, however, can be considered an ISO open standard scene-graph design that is easily extended to offer new functionality in a modular way.

XML [4]. Extensible Markup Language, XML is the standard markup language used in a wide variety of applications. The X3D file format is based on XML, and H3D comes with a full XML parser for loading scene-graph definitions.

OpenGL [5]. Open Graphics Library, the cross-language, cross-platform standard for 3D graphics. Today, all commercial graphics processors support OpenGL accelerated rendering and OpenGL rendering is available on nearly every known operating system.

STL - The Standard Template Library is a large collection of C++ templates that support rapid development of highly efficient applications. It provides also a quite rapid development: by combining X3D, C++ and the scripting language Python, H3D offers three ways of programming applications that gives you the best of both worlds - execution speed where performance is critical, and development speed where performance is less critical.

6.3 Haptics Rendering Engine: HAPI

HAPI is an open-source, cross-platform, haptics rendering engine written entirely in C++. It is device-independent and supports multiple currently available commercial haptics devices. This means that the application can be written just once and no code needs to be modified to use another device. It gives the possibility to choose between different rendering algorithms, different force effects and several kinds of surfaces to create the feeling that you want or create your own custom made effects. HAPI has been designed to be highly modular and easily extendable.
Furthermore, HAPI is integrated in the H3DAPI. This make the development faster, since both graphic and haptic rendering can be achieved by building just one scene-graph.

As other alternatives to the use of HAPI, there are GHOST or the more recent OpenHaptics,

[2]This version of H3D supports the following devices: Phantom Device, Force Dimension Device and Novint Falcon.
[3]http://www.web3d.org
[4]http://www.w3.org/XML
[5]http://www.opengl.org

both developed by SensAble. Ghost is the original API and uses graph structure, but does not perform any graphical rendering (it has callbacks to allow users own graphical rendering). OpenHaptics is the SensAble newest solution for haptic rendering. It is more powerful than GHOST, but since it is not based on graph structure it can be more difficult and not so quick to work with. As GHOST, OpenHaptics does not deal with graphic rendering. Among other things, the OpenHaptics toolkit includes:

- *QuickHaptics*, a micro API that makes it easy to write new haptic applications or add haptic feature to existing applications;
- *Haptic Device API* (HDAPI), which provides a low-level access to the haptic device (enabling to render forces directly offering control over configuring the runtime behavior of the drivers) and provides convenient utility features and debugging aids;
- *Haptic Library API* (HLAPI) provides high-level haptic rendering and is designed to be familiar to OpenGL API programmers. It allows significant reuse of existing OpenGL code and simplifies synchronization of the haptics and graphics threads.

6.4 Framework for software development: Qt

Looking for a framework for developing cross-platform GUI application, the two best solutions seems to be wxWidgets and Qt. To satisfy the objectives, these two C++ frameworks are very similar and offer, more or less, the same possibilities. The opinions on them are very discordant depending on the programmer.

Another point is that using wxWidgets you have to write the makefile by your own (that could be not that easy), whereas in Qt QMake is very simple to use, so there's no need to edit MakeFiles manually; using CMake this is not a relevant difference for this work.

Qt introduces an innovative alternative for inter-object communication, called "signals and slots", that replaces the old and unsafe callback technique used in many legacy frameworks: Qt automatically connects signals to slots based on the names. Moreover, Qt provides a very useful and polished IDE (QtCreator) tailored to the needs of Qt developers. It includes C++ code editor, integrated GUI layout and forms designer, project and build management tools, integrated, context-sensitive help system, visual debugger, rapid code navigation tools and supports multiple platforms.
Both of them provide another very useful tool for graphically designing user interface: wxBuilder for wxWidgets and QtDesigner for Qt.
Furthermore, Qt provides also support for 3D graphics and XML.

WxWidgets is used in many applications and one of these is H3DViewer, the Graphic User

Interface provided by SenseGraphic to load the scene-graphs made using H3D. Since Qt's continuous spreading and other applications in the laboratory are made in Qt, it was thought that Qt would be the best choice also for future integration of this software in those already existing.

Chapter 7

Haptics Interface

The main **objective** of this module is to allow the navigation within a generic bronchial tree using a haptic device as if it were a bronchoscope, seeing the same scene and feeling the same sensations as if the user were carrying out a real bronchoscopy. The idea is to allow the user to move a camera (setting the orientation and position) with the same movements as if he were handling a real bronchoscope. This means finding a good correspondence between the movements that the doctor has to make with the bronchoscope to explore the bronchial tube and the movements available with the Phantom Omni haptic device.

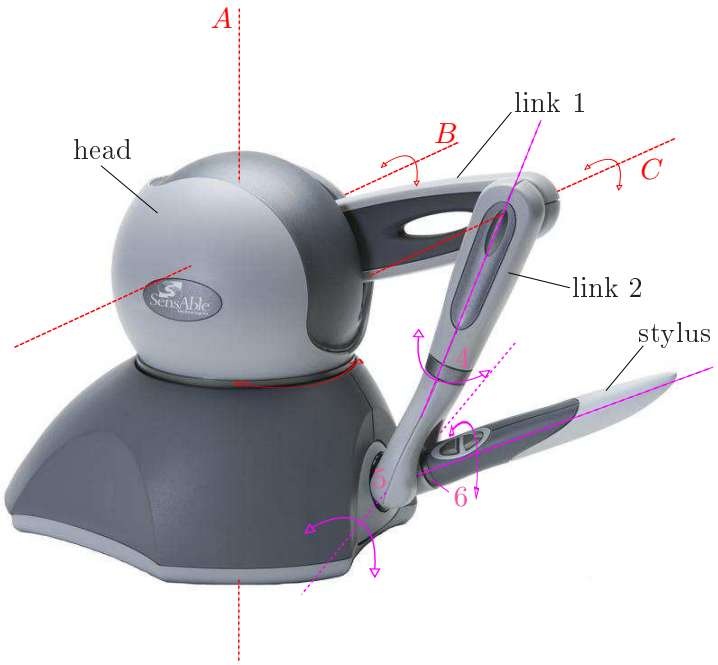


Figure 7.1: Phantom Omni with the axis of movement.

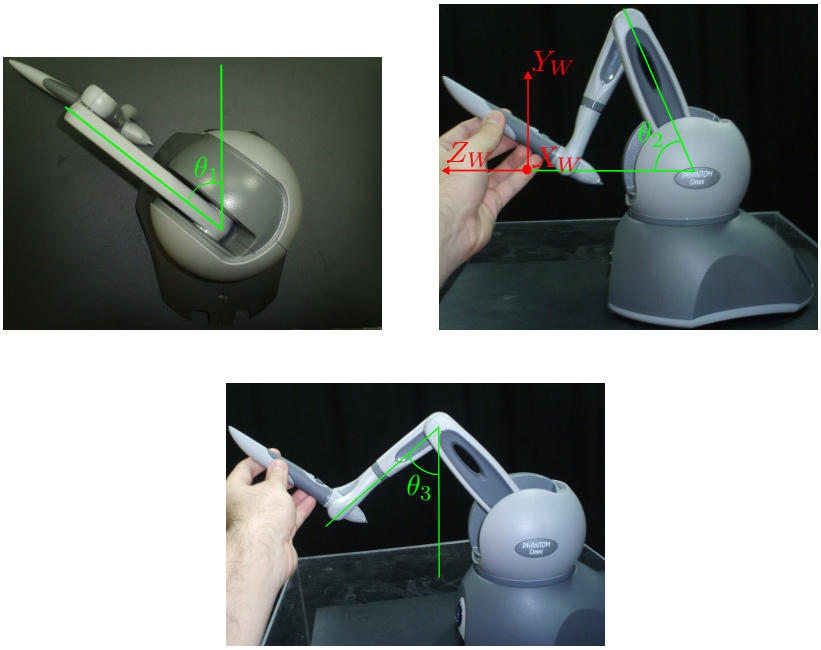


Figure 7.2: Angles of the joints of the Phantom Omni. The red frame represents the origin of the device world coordinates.

Before describing the main features of the navigation and how the Phantom Omni is used for simulating the bronchoscope, it is necessary to specify some global references of the device itself. As it can be seen in figure 7.1, the Phantom Omni has a pen-like tip called *stylus*, which is fixed to the base by two links and a round *head*. This part rotates around the A axis (θ_1), while the first link rotates around B (θ_2) and the second link around C (θ_3) axis which is parallel to B (fig. 7.2). The last three joints (*gimbal elements*), marked with numbers 4, 5 and 6, allow the stylus to rotate around the orthogonal axis system centered in the joint 5, and the consequent orientation (with respect to the world coordinates (X_W, Y_W, Z_W)) can be read from the device.

The endpoint location of the physical haptic interface as sensed by encoders is called *Haptic Interface Position* (HIP). In proxy-based rendering algorithms (such as those included in HAPI and described in section 5.3), the HIP (and the whole haptics device with it) has a virtual representation called *proxy*. The proxy follows the position of the device, but it cannot pass through the objects' surfaces: when a shape is touched, the proxy stays on the surface, even though the haptic device actually has penetrated the surface. Forces are then generated to bring the haptic device out of the surface towards the proxy. When the user moves the haptics device the proxy follows the movement but on the surface. The kind of feedback forces, the movement of the proxy and, therefore, the way the surface feels can be controlled by HAPI. When a surface is touched we can consider that the virtual point representing the HIP will divide into two: the proxy that stays on the surface and

the point that represents the real device position (penetrating the surface) which is called *probe* (or tracker). Where no collisions are detected the probe and the proxy coincide. Even if the proxy can assume many shapes, the real proxy, i.e. the element that interact with virtual objects, is a point, except for Ruspini's Rendering algorithm, where the proxy is a sphere.

In the default situation, using H3D, the device position and orientation are identically transmitted to the virtual world, so the probe (or tracker) position and orientation coincide with the device position and orientation, respectively. H3DAPI supplies the user with two elements (a matrix and a rotation vector) that can modify the way the device position and orientation are "translated" in the virtual world. These two elements are *positionCalibration* and *orientationCalibration*[1].

positionCalibration: It is a field that accepts a `Matrix4f`[2], a 4×4 matrix which has the form of a transform matrix:

$$T_{calib} = \left[\begin{array}{ccc|c} & R & & \begin{matrix}P_x\\P_y\\P_z\end{matrix} \\ \hline 0 & 0 & 0 & s \end{array}\right]$$

where R defines a rotation, P_x, P_y, P_z, define a translation vector and s a scale factor. It works as follows:

$$P_{probe} = T_{calib} \cdot P_{device}, \text{ with } P_{device} = \begin{bmatrix} P_{x_{dev}} \\ P_{y_{dev}} \\ P_{z_{dev}} \\ 1 \end{bmatrix}$$

where $P_{x_{dev}}$, $P_{y_{dev}}$, $P_{z_{dev}}$ define the device position as obtained by the encoders. The important thing to underline here is that `positionCalibration` influences just the way the device position is translated in the virtual world and does not affect the orientation. The rotational part of the matrix (if it is not the identity) will transform a device translational movement in $+X$, for example, in a movement of the virtual HIP that does not occur on a horizontal line. It will be, instead, a movement on a straight line oriented as described by the rotational part, corresponding to the virtual world X-axis. Moreover, this rotational part does not influence the way the device orientation is translated into the virtual world. For example, if the viewpoint orientation were associated to the one of the stylus, the `orientationCalibration` were the identity (default situation) and the user were handling the stylus so as to have the same orientation as the world, the loaded scene will show the world in the normal position: Y-axis pointing upward, X-axis pointing to the right and the Z-axis pointing out of the screen.

[1] positionCalibration and orientationCalibration are two fields of the node PhantomDevice, which is a node of the H3DAPI.

[2] Matrix4f is a type defined in H3D: it is a 4×4 matrix whose elements are floating-point numbers

orientationCalibration: It is the field that provides the calibration of the orientation. This field accepts objects of type `Rotation`, which is a vector defining an orientation in a angle-axis way. This vector has four elements: the first three describe the axis around which the rotation is done and the fourth element is the angle in radians. In H3D Rotation is also a function that takes as input a `Matrix3f`[3] and returns a Rotation vector. Considering `orientationCalibration` a matrix (called R_{calib}), it works as follows:

$$R_{probe} = R_{calib} \cdot R_{dev} \tag{7.1}$$

where R_{probe} is the orientation of the probe and R_{dev} is the orientation of the device as it is read by the encoders.

As said for `positionCalibration`, `orientationCalibration` only affects the way the device orientation is translated into the probe orientation. In other words, if the device is moved in a $+X$ direction and its `positionCalibration` is the identity matrix, then the probe will translate along the world X-axis (irrespective of the values of `orientationCalibration` R_{calib}).

7.1 Bronchoscope Modeling

As seen before, in a bronchoscope we can distinguish three degrees of freedom (fig. 2.1): forward-backward movement, the rotation along the tube central axis and the rotation of the tip of a $\pm 90^o$ range. The first and the second actions are performed by the doctor acting directly on the bronchoscope handle and tube, pushing and pulling or rotating the bronchoscope tube itself through the bronchial tree. The tip can be rotated just in one plane whose orientation depends on the orientation of the tube. The tip rotation is provided by a wheel (element (B) in the fig. 2.1) that can be rotated till the tip reaches an orientation of $\pm 90^o$ with respect to the bronchoscope D axis.

Considering how the Phantom Omni is handled by the user and wanting to reproduce the movements of the bronchoscope (as much similar as possible), it has been decided to adopt the following correspondence between the three basic movements:

- The forward-backward movement made by pushing or pulling the tube will be performed by translating the device tip along the device Y-axis. A negative value will correspond to a forward movement, while a positive value will correspond to a backward one.
- The rotation along the tube central axis will correspond to the rotation of the 6^{th} joint of the device.

[3] A 3×3 defining a rotation whose elements are floating-point numbers

- The orientation (between $+90^o$ and -90^o) of the camera positioned on the tip of the bronchoscope is obtained acting on the 5^{th} joint, but (as explained further in section 7.2.1) it will not be 5^{th} joint angle θ_5.

7.2 Navigation System

The basic idea is to position the camera in the proxy and make it move with it. The movements done with the device to make the camera move will be as much similar as possible to the ones done when handling a real bronchoscope.

As explained in chapter 6, H3D has been used both to describe the graphic scene and to render the haptics properties. The Scene is described with a XML-like graph (X3D file). The node `<Scene>` is the "highest" node in a X3D graph (no other X3D node can contain `scene`) and contains all the other nodes describing the shapes and haptics properties. To render the scene, a window inheriting from `H3DWindowNode` must be created. To this kind of window, whose implementation will be expounded later on, in chapter 8, a viewpoint can be associated, which represents the camera of the H3D scene. This `Viewpoint` node inherits from the class (which represents a node) `X3DViewpointNode`, which, in turn, contains two fields named "position" and "orientation". They specify, as the names tell, the position and the orientation of the viewpoint, respectively. The viewpoint node has a frame (coordinate system) associated to it. Per default the viewer is on the Z-axis looking down the $-Z$-axis toward the origin with $+X$ to the right and $+Y$ straight up (fig. 7.3).

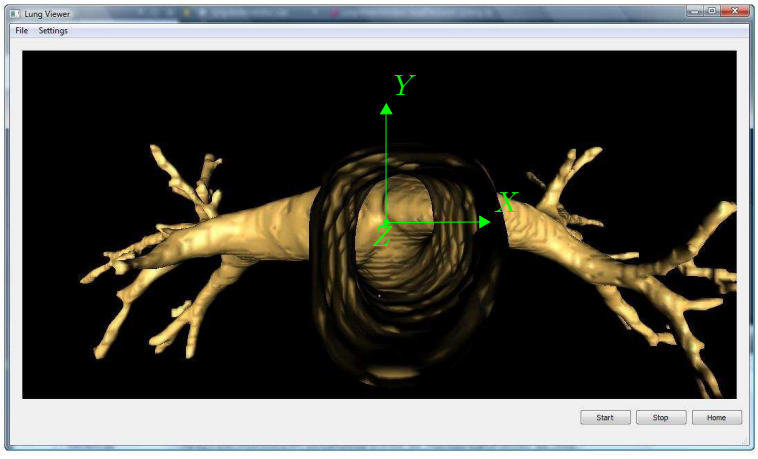
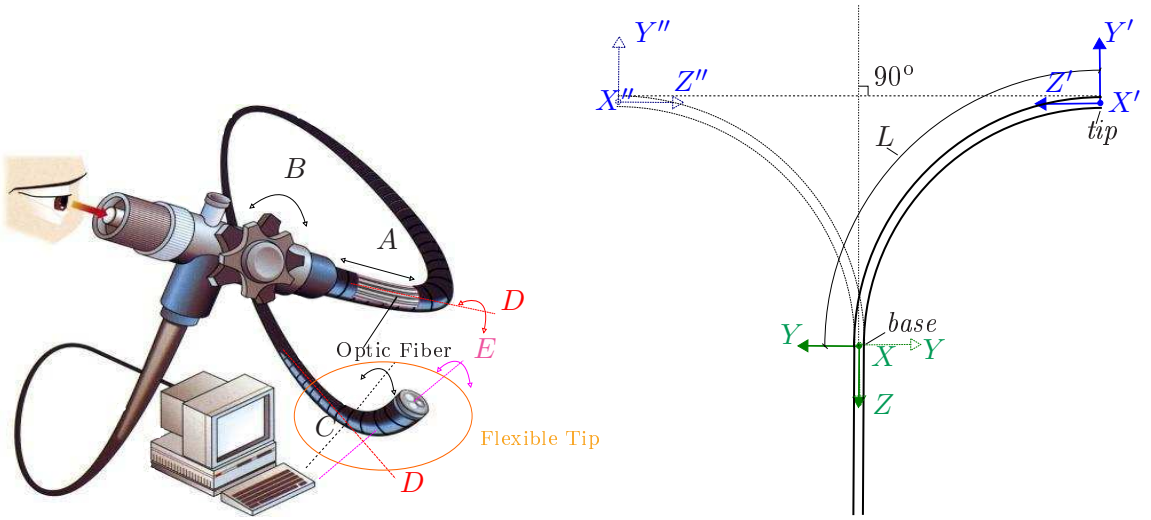


Figure 7.3: Default viewpoint orientation drawn on the application main window.

The configuration of the bronchoscope tip is defined by a position and an orientation. Trying to use the haptic device as a bronchoscope, there are two possible ways to set the tip orientation with



(a) *Drawing of a real general endoscope showing the DOF and the axis of rotation.*

(b) *Simplified representation of the bronchoscope tip. The frame* (X'', Y'', Z'') *is the same as* (X', Y', Z') *when the tip is turned in the opposite direction* $(+90°)$.

Figure 7.4: Degrees of freedom and reference frames in the considered model of bronchoscope.

the device movements: one is controlling directly the camera on the tip (described in section 7.2.1) and the other is controlling the camera from the base of the tip (described in section 7.2.2), that is the beginning of the flexible bronchoscope final part (that one that can be moved from -90^o till $+90^o$, see figure 7.4). Both of them have been implemented and the final decision about which one is the best will be let to the doctors who will test the application.

Looking at figure 7.4 it can be better seen how the bronchoscope is modeled and it can be better understood what each of the implementations are referring to. Figure 7.4(b) shows the two frames associated to the bronchoscope tip which the two different implementation of camera controlling refer to. The navigation implemented controlling directly the camera (section 7.2.1) will refer to the blue coordinate system (X', Y', Z'), whereas the other implementation (section 7.2.2) will refer to the green one (X, Y, Z). Furthermore, between the two images (7.4(a) and 7.4(b)) some correspondences can be underlined (leaving out the orientation):

- The D-axis coincides with the Z-axis whereas the E-axis corresponds to the Z'-axis. This two axes have the same orientation only if the tip is totally extended.

- C-axis refers to X-axis and corresponds to the beginning of the bronchocope flexible tip which can be rotated, as already mentioned, in a range of $\pm 90^o$. In other words, it marks the beginning of the tip which the wheel actuates on.

The two possible implementations mentioned have, basically, the same process and differs only in few things regarding how the calibration matrices are calculated and how the device controls the camera. The internal cycle that allows navigation can be described as follows:

```
1 Haptics_loop(){
2
3         Read_device_info(); // read the position and the orientation of the device
4         Compute_controls(); // Dz: bronchoscope advance, angles of the rotations to
              apply to the camera
5         Update_viewpoint();
6         Update_calibration();
7         Compute_forces();
8
9 }
```

7.2.1 Moving directly the camera

The first possible implementation of the "Haptics_loop()" is using the haptic device movements to control directly the position and orientation of the camera.

Rotation Movement

When asking the device for the orientation, the device returns a `Rotation` object which can be easily converted to a 3×3 matrix using the functions of H3D. This matrix defines a rotation between the fixed frame of the device and the frame associated to the HIP ((X_W, Y_W, Z_W) and (X, Y, Z) frames of picture 7.5, respectively), which depends on the joints 4, 5 and 6 (fig. 7.1).

Exploring a scene with a flying camera controlled by an external device can present some difficulties. The problem is that the device reads the values of position and orientation with reference to his physical world frame which, in a default situation, coincides with the scene world frame. So, when the camera and the scene world frame orientations are not the same, the camera will not respond to the device input as expected: when the camera is rotated around Z-axis, for example, its top stops coinciding with the top of the scene world. In this case, if the observer wants to look upward (with respect to what he is looking), he will spontaneously move the stylus tip up but this movement will not be translated to the camera looking upward, because it corresponds to an upward movement (positive rotation along X) with respect to the scene world coordinates. This problem is accentuated when the 3D scene has no reference points to distinguish which are the scene top and bottom or what is right and left. A bronchial tree exploration can be considered such a scene, because of the cylindric shape of bronchus and the symmetry of the lungs. In these cases

it is important to update the calibration matrices in order to translate the device movements into the expected camera movements. That can be done by applying the "infinitesimal" variation in the device orientation (every clock signal) to the rotation of the actual camera frame.

Before continuing, a clarification has to be made on the meaning of the joint angles. The values read by the encoders (called gimbal angles) are the absolute angles of the joints but do not define the device orientation (θ_5 and θ'_5 in figure 7.5). The difference can be seen in figure 7.5: doing a translation in world Y-axis (where only the device position is supposed to change, but not its orientation) will modify the gimbal angle of the 5^{th} joint but the orientation, correctly, will not change. From now on, if no other thing is specified, when it will be talking about changes in the 5^{th} or 6^{th} joint, it will be referring to changes that cause a variation in the device orientation.

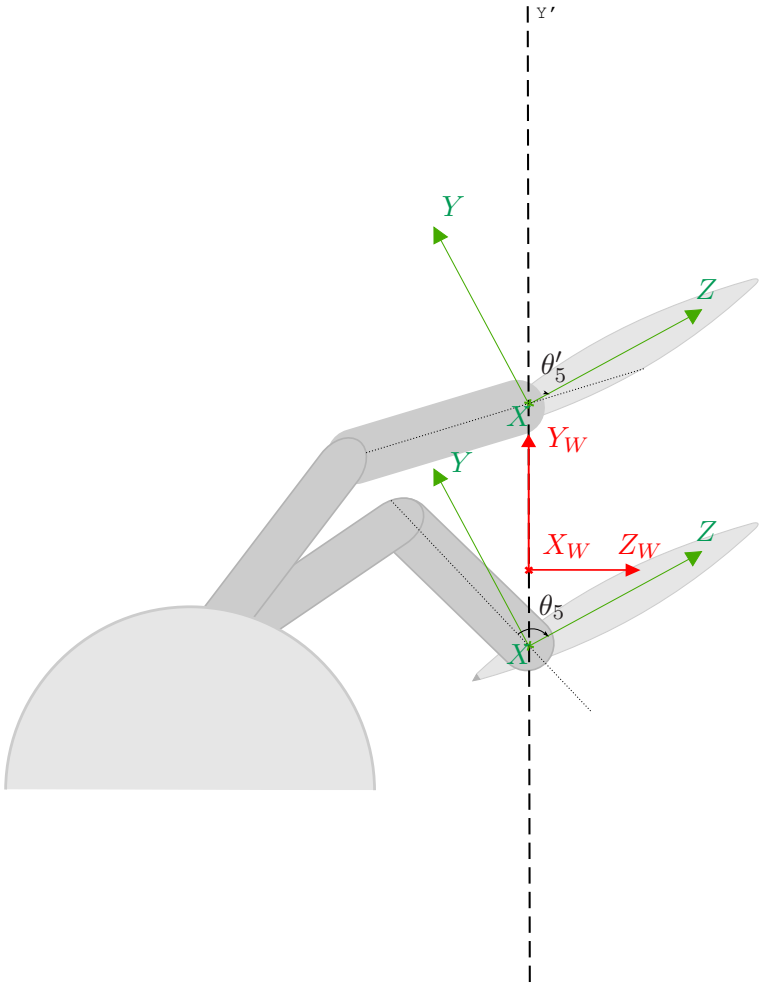


Figure 7.5: Visual representation of the difference between the gimbal angles (θ_5) and the device orientation. In a $+Y$ translation change the value of the gimbal angles but not the orientation.

The correspondence between movements of the device and those of the camera has to take into consideration the DOF of the real bronchoscope. That being so, the rotation of the 6^{th} joint can

intuitively correspond to a camera rotation around its Z-axis (that is the viewpoint Z-axis). A clockwise rotation of 6 will correspond to a clockwise camera rotation around its central axis. The 5^{th} joint can be used as the wheel of the real bronchoscope, moving the camera up and down with respect to the camera frame. With regard to the 4^{th} joint, a real bronchoscope has not this movement so the value from this joint will not be considered and will not affect the camera orientation.

Summarizing all what has been told, the aim here is that a rotation in the 5^{th} joint always corresponds to a rotation about the camera X-axis and that a rotation in the 6^{th} joint always corresponds to a rotation about the camera Z-axis. These two read angles do not correspond directly to the device orientation about X and Z-axis, because a rotation of the 5^{th} affects the orientation depending on the position of the 6^{th} one. Figure 7.6 explains better this issue. The (X, Y, Z) frame is the one related with the camera and (X_W, Y_W, Z_W) frame is the world frame, which has been drawn twice with the origin coincident with that of the HIP to see better the orientation which relates the two reference systems. It can be useful to remember that the camera looks towards $-Z$, with $+Y$ pointing up and $+X$ pointing right. In case 1 the movement of 5 change the orientation of the camera up and down, that is a rotation around the camera X-axis. When the joint 6 is rotated -90^{o}, the rotation of 5 provokes a rotation around the Y-axis, which means a right-left changing in the orientation of the camera, which is not the expected behaviour.

A solution to this problem is to calculate the orientation of the device as a matrix and from this calculate the rotation angles with respect to the fixed axis. After that, it will be sufficient to use the rotation angles around the X_W and Z_W to rotate the camera frame (which is associated to the probe) around its current X-axis and Z-axis, respectively.

To achieve this, the first step is finding the angles with respect to the world axis. Defining the rotation matrix:

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} = R_x(\varphi)\, R_y(\vartheta)\, R_z(\psi),$$

what is wanted to find are the values of the rotation angles ψ and φ around Z and X axis, respectively. These are:

$$\varphi = -\arctan 2(r_{12}, r_{11}) \tag{7.2}$$

$$\psi = -\arctan 2(r_{23}, r_{33}) \tag{7.3}$$

where $\arctan 2(y, x)$ is the angle in radians between the positive X-axis of a plane and the point given by the coordinates (x, y) on it.

When the device is asked to return its orientation, it returns a rotation matrix of this kind:

$$R_{dev} = R_x(\gamma)\, R_y(\varsigma)\, R_z(\alpha),$$

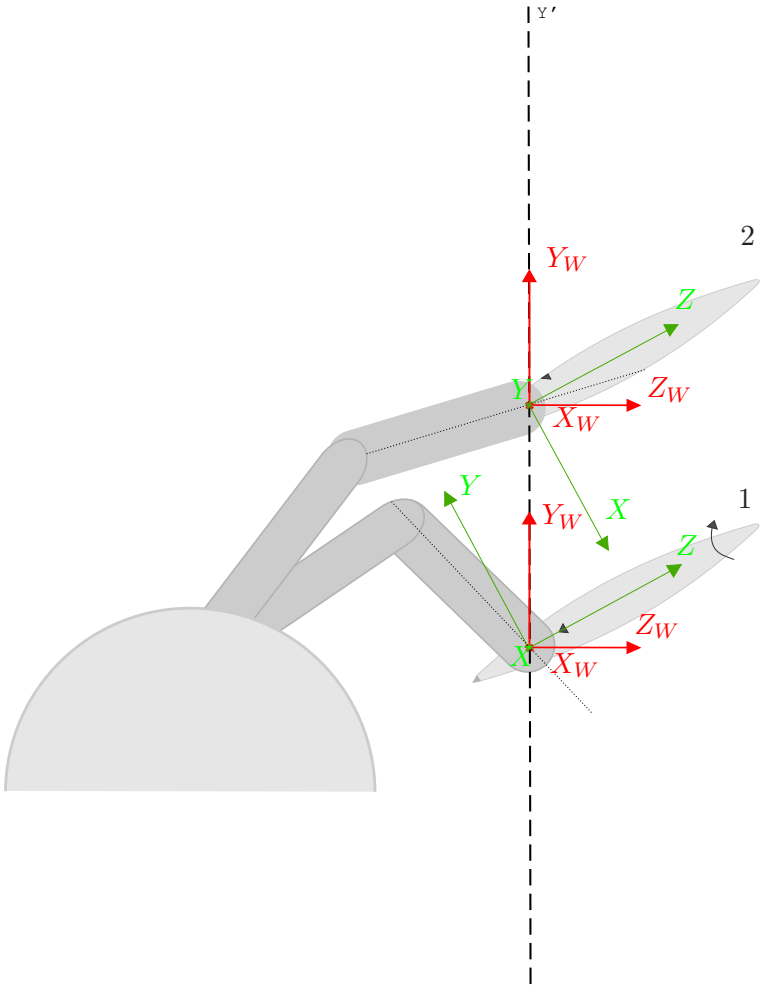


Figure 7.6: The effect of the joint values on the orientation. When the 6^{th} is rotated -90^o the 5^{th} joint will rotate around the world X-axis (the red one) which corresponds to the HIP Y-axis.

It is possible to apply (7.2) and (7.3) to get the angles γ and α, which define the orientation of the HIP with respect to the world coordinates[4].

Once these angles values are found their variations have to be calculated to rotate the camera in the right way. So, when one of these angles changes by an infinitesimal Δ[5], this rotation of Δ along one axis will be applied to the actual camera frame with respect to the corresponding axis.

There are other features to take into account. One important point is the 5^{th} joint which controls the rotation of the wheel, rotating the camera 90^o up or 90^o down (around the C-axis of the figure 7.4(a)). When the doctor wants to turn right or left, he has to orientate the bronchocope

[4]The HIP reference system is left-handed system with the X-axis opposite with respect to the one drawn here. For reasons of clearness, it has been decided to use a right-handed frame for the modeling changing the sign of α in the implementation.

[5]With the term infinitesimal here, it is meant the variation of the angle occurred during a clock cycle.

in order to position the plane, which the tip rotates on, in a horizontal position. Then, he actuates on the wheel to turn the tip and push the tube forward to enter in the desired bronchus. Once got to the right path, he combines the rotation of the wheel, necessary to straighten the bronchocope tip, with pushing the tube so as to go straight ahead. To render this action of the doctor that puts the wheel back to default position once entered the right bronchus, in this kind of implementation (using the device to control directly the camera), it has been decided to rotate the camera on this plane by a rate control. The rotation angle about camera X-axis (γ) is increased (or decreased) by a quantity equal to the difference between the current device angle and a reference angle previously set. Considering the way the bronchoscope is handled (see figure 2.3), this reference angle was set to 45^{o}.

Implementing all these features, the movements made during the exploration will better trace the real movement. In fact, when the user wants to turn right or left because he gets to a fork, he has to turn the sixth joint till he obtains the two bronchial tubes in a vertical plane, then he moves the HIP up or down (depending on which bronchus he wants to explore) rotating the 5^{th} joint, till the camera gets to the needed orientation to enter the bronchus. Subsequently, he has to put the stylus in the reference position to make the camera stop rotating and be able to go straight, as if he was rotating the wheel back to the original position.

In light of what has been shown up to this point, the wanted orientation calibration matrix is the result of several infinitesimal rotations applied to an accumulated rotation matrix. In other words, the orientation matrix at instant i is:

$$R_i = R_{i-1} \cdot R_z(\Delta\,\alpha) \cdot R_x(\Delta\,\gamma), \tag{7.4}$$

where

R_{i-1}	is the orientation matrix of the camera at instant $i-1$
$\Delta\,\gamma = \gamma_i - \gamma_{i-1}$	and
$\Delta\,\alpha = \alpha_i - \alpha_{i-1}$	.

Considering the way γ is incremented, $\Delta\,\gamma$ can be written as

$$\Delta\,\gamma = \pi/4 - \gamma_i.$$

Since the aim is also to interact with the virtual scene the observer is looking at and, for example, touch the bronchus, the proxy has to be in the same position and same orientation as the viewpoint. With regard to the orientation, this means setting the `orientationCalibration` field in order to obtain in (7.1):

$$R_{probe} = R_i$$

this can be made by setting:

$$R_{calib} = R_i \cdot [R_{dev}]^{-1}\,. \tag{7.5}$$

This makes the probe (and so the proxy, if there are no collisions) and the viewpoint to have the same orientation.

Finally, it has to be taken into consideration also the fact that the bronchoscope tip cannot bend more than 90º in both sides (fig. 7.4). Every time the angle changes, the application will control that the rotation angle around the local X-axis never exceeds $\pm 90°$.

Insertion Movement

The insertion movement implies a modification of the position of the camera (positioned on the tip of the tube), which can be considered the viewpoint of the real scene. Imagining the real camera with the same frame of the viewpoint, the pushing movement causes a translation of the tip towards the camera $-Z$-axis. So, knowing the orientation of the camera, it is possible to translate the viewpoint in the correct direction. To make the viewpoint move following the haptic device movements, it is changed with the position of the proxy.

A consideration about the workspace has to be done, before describing in details the implementation of this part. As it has been showed in previous sections, there are several kinds of workspace. When building a haptic display system, the optimal system is the one which is able to make the application workspace (AW)[6] coincide with that zone in which the haptic device provide its better performance.
In this case, the application workspace is much bigger than the nominal (NW) and the real workspace (RW), so something is needed to reach and be able to navigate all the AW. There can be two solution: use a mouse-jump to translate the NW in another volume space of the AW or use the haptic device to establish a velocity value and translate the probe and the NW where needed, in the application volume.
The second solution has been chosen, because this one seems to be more comfortable compared to the hundreds of mouse-jumps needed to explore the whole bronchial tree.

The insertion movement is provided by moving the stylus up (backward) and down (forward) the Y-axis. At every clock cycle, the device is asked to read the position value on the Y-axis. This read value is taken as the linear velocity of the viewpoint (VP): if it is negative the viewpoint will translate in its $-Z$-axis direction and if it is positive it will go backward (in the $+Z$-axis direction). In other words, the device Y position corresponds to an increment (Δz) in the camera Z-axis.
This behaviour is performed by passing (every clock loop) to the VP position the tip position (where

[6] If determined virtual simulation is defined, it is the "virtual" volume used by the application

the camera is placed) calculated as follows:

$$P_{camera_i} = P_{camera_{i-1}} + R_{probe} \cdot \begin{bmatrix} 0 \\ 0 \\ \Delta z \end{bmatrix} \tag{7.6}$$

where R_{probe} is the tracker (or probe) orientation, Δz is the linear increment calculated from the device as explained before and P_{camera} is the position of the camera, which coincides with the one of the tip. Since the camera position and the probe position have to coincide, the `positionCalibration` matrix has to be set to:

$$T_{calib_i} = \left[\begin{array}{ccc|c} & I & & P_{camera_i} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \cdot \left[\begin{array}{ccc|c} & I & & P_{dev_i} \\ \hline 0 & 0 & 0 & 1 \end{array}\right]^{-1} \tag{7.7}$$

where P_{camera_i} and P_{dev_i} are the camera and device position vector at instant i, respectively.

The real bronchoscope can only translate along its longitudinal axis, since, once inserted, the trachea or the bronchus do not enable movements on the plane perpendicular to the bronchoscope tube. To render also this characteristic, the device movement along its X-axis has been blocked by sending it a force proportional to the device X-coordinate so as to bring the device to the X-origin.

To make the procedure clearer, the "Haptics_loop" can be summarized as follows (in pseudo-code):

```
   Read_device_info()
{
   P_device=device_position
   R_device=device_orientation
}


   Compute_controls()
{
   Δz =P_device.y
   α_i =rotZ(R_device)
   γ_i =rotX(R_device)
   Δα_i = α_i − α_{i−1}
   Δγ = π/4 − γ_i
}
```

```
Update_viewpoint()
{
```

$$R_{camera_i} = R_{camera_{i-1}} \cdot R_z(\Delta\,\alpha) \cdot R_x(\Delta\,\gamma) \;//\text{ from eq. (7.4)}$$

$$P_{camera_i} = P_{camera_{i-1}} + R_{probe} \cdot \begin{bmatrix} 0 \\ 0 \\ \Delta\,z \end{bmatrix} \;//\text{ from eq. (7.6)}$$

```
}
```

```
Update_calibration()
{
```

$$R_{calib} = R_i \cdot [R_{dev}]^{-1} \;//\text{ from eq. (7.5)}$$

$$T_{calib_i} = \left[\begin{array}{ccc|c} & I & & P_{camera_i} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \cdot \left[\begin{array}{ccc|c} & I & & P_{dev_i} \\ \hline 0 & 0 & 0 & 1 \end{array}\right]^{-1} \;//\text{ from eq. (7.7)}$$

```
}
```

```
Compute_forces()
{
    // Constraint along x-axis
    // send to the device a force proportional to its X-position
    computeFx(P_device.x)
    // if the surface node perceives a contact, it sends a force
    //along device Y-axis to push the camera backward
    if (surface.isTouched()) compute_Fy()
}
```

7.2.2 Moving the camera from the base of the tip

The way of controlling the camera described in the last section is easy and intuitive but do not correspond faithfully to the real bronchoscope behaviour: for example, when the doctor bends the tip of the bronchoscope and turns the bronchoscope around the tube central axis, he will not observe the image rotating around the camera Z-axis but about the D axis (fig. 7.4(a)). In other words, if the camera is 90º bent, he will observe the bronchus wall and rotating the bronchoscope handle he will move the camera from the bottom to the top, always looking at the bronchial wall.
Translating this question to the device, it means that a rotation of 6 correspond to a rotation around the camera Z-axis, just in the case the E and D axis were aligned.

To have a better correspondence with the real bronchoscope movements, the moving of the camera from the base is proposed. This requires a different model of the bronchoscope. The final bronchoscope part has been modeled like a kinematic chain composed by n links of the same length (fig. 7.7). It actuates like a robot which has the camera in its TCP[7].

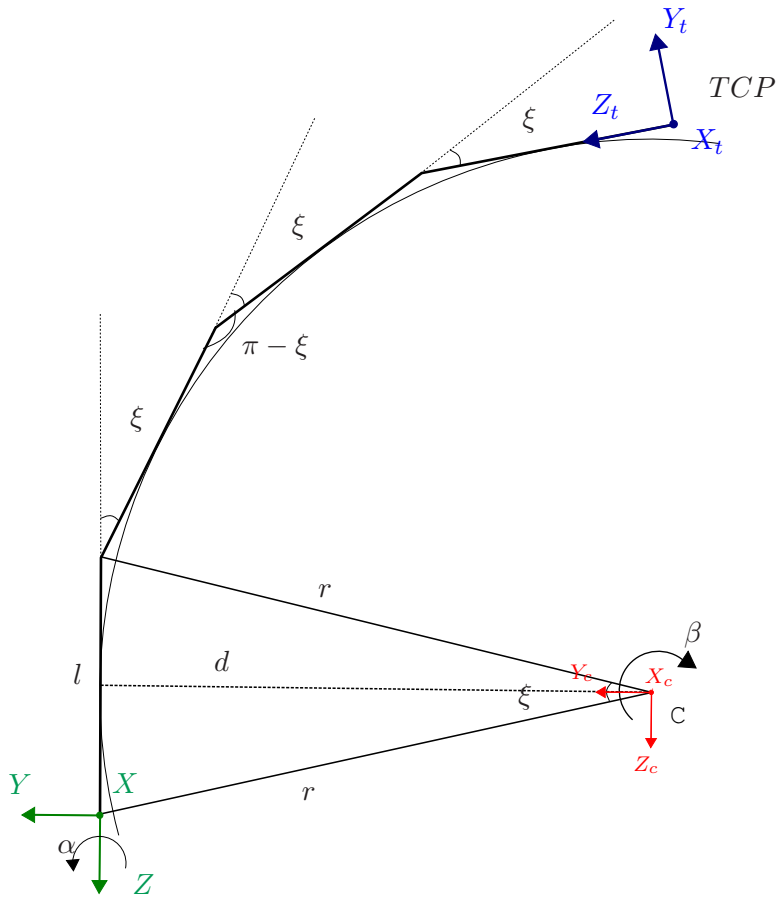


Figure 7.7: Modeling of the bronchoscope for $n = 4$.

[7] Tool Center Point

Rotation Movement

The angle of the joint 6 of the device determines α and joint 5 will modify the total bending angle of the tip, that is $(n-1)\cdot\xi$.
In this way there is a better correspondence between the handle-tip relation of the real bronchoscope and the device-VP relation of the "virtual" one. The rotation of the joint 6 always corresponds to a rotation around the Z-axis of the green frame (X, Y, Z) (which is the D axis in fig. 7.4(a)) and not around the camera Z-axis (Z_t of figure 7.7). It will happen only if the tip is totally extended, that is $\xi = 0$. The green coordinate system (X, Y, Z) is the one which the device movements are associated to, while the blue one (X_t, Y_t, Z_t) is the one the viewpoint is referred to.

Insertion Movement

Also in this case, the linear increment of the position is determined by the Δy of the device (as described in section 7.2.1). What differs here is how this linear increment (which is a Δz with respect to the (X, Y, Z) frame) is translated to the movement of the virtual tip. It has been imagined that, when the tip is bent by a certain angle, the doctor's intention is to turn into a bronchus by pushing the tube and make the side of the bent tip lean against the bronchus wall and make the bronchoscope tube slide into it. To render this behaviour, every increment in the Z-axis of the (X, Y, Z) coordinate system (fig. 7.7) will be translated in a rotation of a certain angle β around an axis passing through C and perpendicular to the plane where the tip is moving on. This angle β, obviously, depends on the increment Δz and on the curvature ξ. If the tip is totally extended, a pushing movement will correspond directly to a straight forward translation. In other words, referring to the geometric model of the same figure (7.7), if $\xi = 0$ the center C of rotation will be located in the infinity and the Δz will be directly a linear translation along green Z-axis (which will coincide with the blue one Z', in this case).

Calculating Viewpoint and Calibration Matrices

At the beginning, it is shown how the movement of the devices are translated into the movement of the green coordinate system, so if nothing is specified, everything will refer to this system. Also another nomenclature convention has to be established. The transformation matrices in homogeneous coordinates will be expressed with the letter T and will have mainly two forms:

$$T_{(x,\alpha)} \text{ or } T_{(x,y,z)}$$

where the first is a 4 matrix defining a pure rotation along the X-axis (or any other axis indicated) of a generic angle α. The second one is a 4 matrix too and expresses a pure translation

of the generic vector (x, y, z). For example:

$$T_{(x,\alpha)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\alpha & -sin\alpha & 0 \\ 0 & sin\alpha & cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ or } T_{(x,y,z)} = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The rotation of α depends on where the (X, Y, Z) coordinate system is placed and oriented in the space, so, really, it will be a rotation of infinitesimal $\Delta\alpha$ applied every clock loop on the current orientation of the system. Figure 7.8 shows the idea. This transformation can be expressed with the following matrix:

$$T_{(z,\Delta\alpha)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\Delta\alpha) & -sin(\Delta\alpha) & 0 \\ 0 & sin(\Delta\alpha) & cos(\Delta\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

As already told, the 5^{th} joint acts on the bending angle of the tip (as shown in figure 7.9). The total bending angle, which is $(n-1)\cdot\xi$ can be maximum 90^o and will be checked every cycle.

As seen before, if $\xi = 0$ an increment of Δz corresponds to a linear increment in the same direction. When the user will modify the value of ξ moving the 5^{th} joint, this Δz will be converted in a rotation of β around X_c-axis. This transformation can be expressed by the matrix:

$$T_{(x,\beta)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\beta & -sin\beta & 0 \\ 0 & sin\beta & cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Looking at figure 7.10), it can be seen all the process to implement this Δz translation and its final effect of rotating the bronchoscope around X_C axis. To rotate an object or, better said, to rotate the frame associated to the object (in this case (X_b, Y_b, Z_b)) about an axis of another frame, some steps have to be followed:

- Translate the frame (X_b, Y_b, Z_b) to the origin of the other frame $((X_c, Y_c, Z_c))$. In this case the translation will be:

$$T_{(0,-d,-\frac{l}{2})} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -d \\ 0 & 0 & 1 & -\frac{l}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

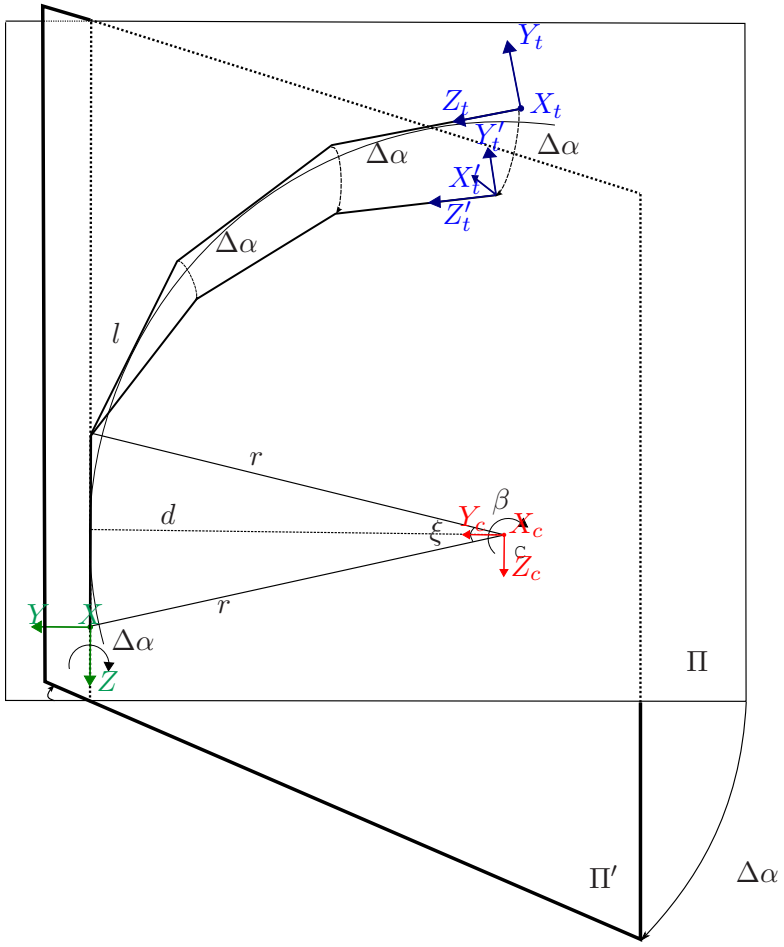


Figure 7.8: Describing the effect of a $\Delta\alpha$ rotation on the movement of the tip.

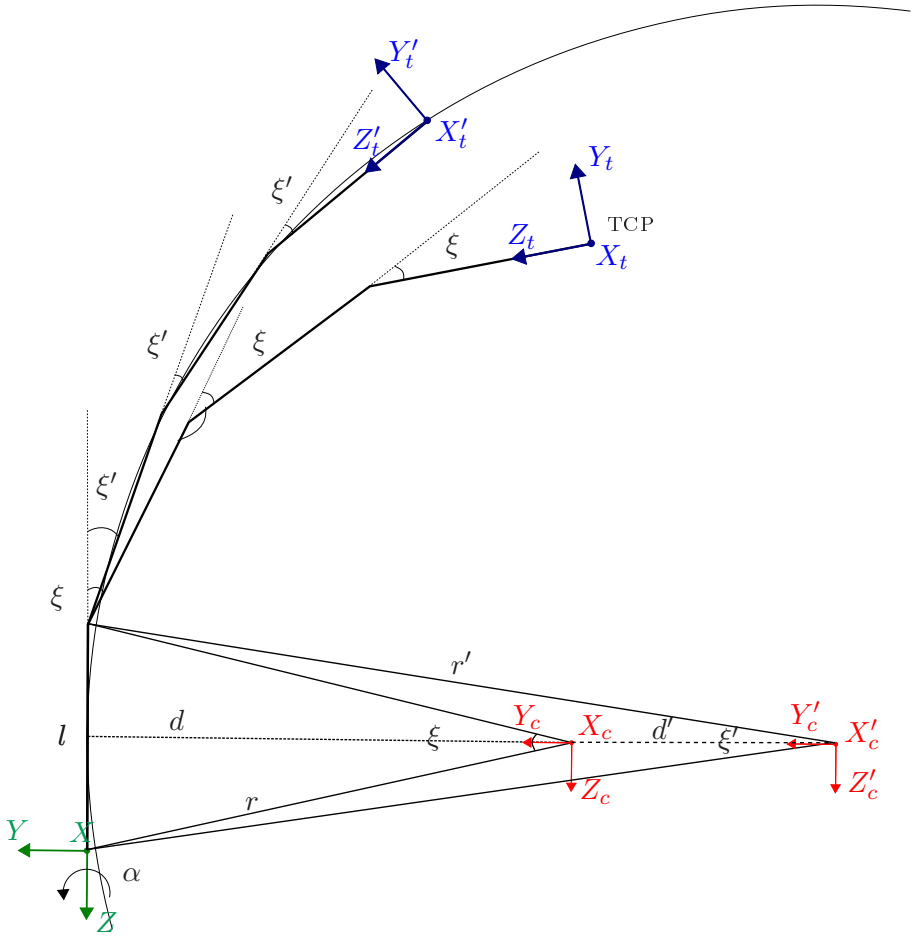


Figure 7.9: Describing the effect of changing ξ on the modeled bronchoscope tip.

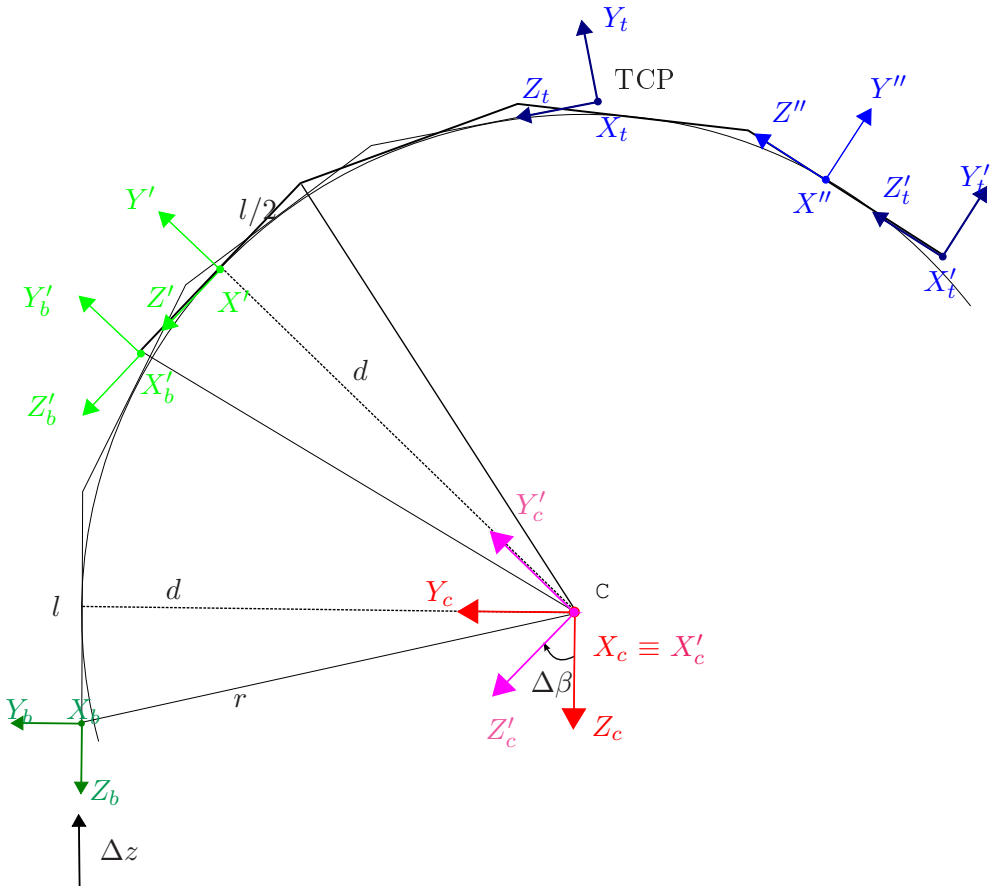


Figure 7.10: Describing the effect of changing Δz on the modeled bronchoscope tip.

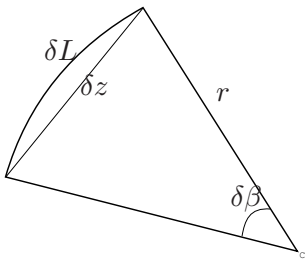


Figure 7.11: In an infinitesimal rotation the arc δL can be indistinct from the cord δz.

- Make the frame to rotate of β about the X_c-axis obtaining (X'_c, Y'_c, Z'_c). Knowing that

$$d = \frac{l}{2} \cdot \frac{cos(\xi/2)}{sin(\xi/2)}$$

and considering that, for infinitesimal rotation, the arc can be indistinct from the the cord δz relative to the arc (figure 7.11), β can be calculated as follows:

$$\beta = \frac{\Delta\, z}{d} = \frac{2\Delta\, z \cdot \sin(\xi/2)}{l \cdot \cos(\xi/2)}.$$

- After having rotated, the inverse of the initial translation has to be applied so as to find the new base position. The new transformation will be a translation of $+d$ along the Y'_c-axis followed by another translation of $+l/2$ along the Z'-axis.

Ascertained that, to actualize the orientation and position of the green base (X_b, Y_b, Z_b), every clock loop the associated transformation has to be actualized as follows:

$$T_{base_i} = T_{base_{i-1}} \cdot T_{(z,\Delta\alpha)} \cdot T_{(0,-d,-\frac{l}{2})} \cdot T_{(x,\beta)} \cdot \left[T_{(0,-d,-\frac{l}{2})}\right]^{-1} \tag{7.8}$$

where T_{base_i} is a 4×4 matrix which defines orientation and position of the base of the tip at instant i.
The real position of the camera and the "touch-point" are on the bronchoscope tip. With this particular modeling, it can be easily derived the orientation and position of the tip with respect to the ones of the base. Since the links have all the same length and every link form the same angle (ξ) with its subsequent, this transformation consists in a rotation of the base frame about the same X_C-axis as before. The rotation angle here is $-n\xi$ and "rotates" the base frame to (X_t, Y_t, Z_t) which is the frame associated to the camera. The same transformation can be expressed as follows:

$$T^{base}_{camera} = T_{(0,-d,-\frac{l}{2})} \cdot T_{(x,-(n-1)\xi)} \cdot T_{(0,d,-\frac{l}{2})}, \tag{7.9}$$

where the rotation angle is $-(n-1)\xi$ and the last translation makes the frame (X'', Y'', Z'') to translate along its Z'' axis till coinciding with (X_t, Y_t, Z_t).

From this last operation, the following result can be easily obtained:

$$T_{camera} = T_{base} \cdot T_{camera}^{base}. \tag{7.10}$$

T_{camera} is a 4×4 matrix of the form:

$$\left[\begin{array}{ccc|c} & R_{camera} & & P_{camera} \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

where R_{camera} and P_{camera} are respectively the orientation and the position of the camera with respect to the world frame. In order to obtain the probe in P_{camera} and with the orientation R_{camera}, it is only needed to make `positionCalibration` and `orientationCalibration` fields to take these two values:

$$R_{calib} = R_{camera} \cdot [R_{dev}]^{-1}, \tag{7.11}$$

$$T_{calib} = \left[\begin{array}{ccc|c} & I & & P_{camera} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \cdot \left[\begin{array}{ccc|c} & I & & P_{dev} \\ \hline 0 & 0 & 0 & 1 \end{array}\right]^{-1}. \tag{7.12}$$

The viewpoint is placed and oriented with the same values as the probe and proxy. The VP position can be kept a bit behind the proxy so as to totally prevent the VP from going out the bronchus wall even in case of collision. If the viewpoint were associated to the probe, in the possible collisions the viewpoint would have followed the probe which penetrate the surface which is not possible for the real bronchoscope tip.

In this implementation, the "pseudo-functions" *Read_ device_ value(), Update_ calibration()* and *Compute_ force* of the "Haptics_loop()" algorithm are identical to those of the other implementation (7.2.1), whereas the central body of the algorithm changes. It can be schematized like this:

```
   Compute_controls()
{
   Δz =P_device.y
   ξ =rotX(R_device)
   α =rotZ(R_device)
   Δα_i = α_i − α_{i−1}
}


   Update_viewpoint()
```

{

$$\beta = \frac{\Delta z}{d} = \frac{2\Delta z \cdot \sin(\xi/2)}{\cdot l \cdot \cos(\xi/2)}$$

$$T_{base_i} = T_{base_{i-1}} \cdot T_{(z,\Delta\alpha)} \cdot T_{(0,-d,-\frac{l}{2})} \cdot T_{(x,\beta)} \cdot \left[T_{(0,-d,-\frac{l}{2})}\right]^{-1} \text{ // from eq. (7.8)}$$

$$T^{base}_{camera} = T_{(0,-d,-\frac{l}{2})} \cdot T_{(x,-(n-1)\xi)} \cdot T_{(0,d,-\frac{l}{2})} \text{ // from eq. (7.9)}$$

$$T_{camera} = T_{base} \cdot T^{base}_{camera} \text{ // from eq. (7.10)}$$

$$R_{camera} = \texttt{rotation}\ (T_{camera})$$

$$P_{camera} = \texttt{translation}\ (T_{camera})$$

}

In the implementation of section 7.2.1 the rotational and translational part of the matrix T_{camera} can be calculated separately, this eases the calculation itself and even the amount of used memory, when storing the data. On the other hand, this implementation requires a more complicated process to find T_{camera}, so the application first calculates the matrix and then takes its rotational and translational part separately.

7.2.3 Going Backward

The backward movement is provided by pulling up the stylus till a position with a positive component along device Y-axis. When the doctor pulls out the bronchoscope from the bronchial tube, the camera go along the same route it did going forward (forced by the fact that it is linked to the bronchoscope flexible tube). To render this way of acting, while going forward in the exploration, the tip orientation and position are memorized in a C++ `vector`, creating like an historical memory of the done path. In the implementation 7.2.1 will be two vectors: one memorizing the viewpoint (VP) position (`Vec3f` elements) and the other the orientation (`Rotation` elements). In the implementation 7.2.2 the vector will be only one memorizing Matrix4f elements which describe the total transformation of the tip with respect to the world coordinates.
When the device reveals a positive value in its Y-axis, the viewpoint fields that were connected to the device, start reading the values from the vector (or vectors) in question. The main thread keeps on popping values from the last position of this vector and assign them to the correspondent field of the viewpoint. In this way, the VP will travel through the memorized path towards the starting point and the user will observe the same scene he observed while going forward.

In the configuration described in section 7.2.1, when the user decides to pass from going backward to go forward and the orientation is not the same a jump may occur in the viewpoint.

The situation can be seen in figure 7.12: the user is going backward and the memorized camera

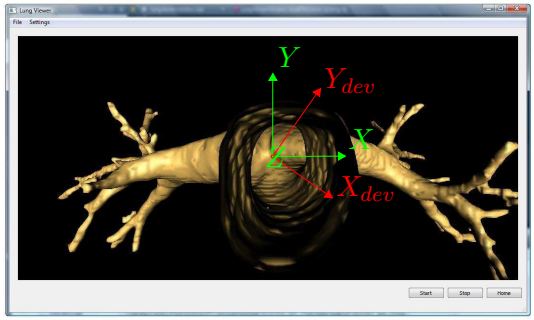


Figure 7.12:

frame he gets to is positioned like (X, Y, Z). The problem comes out when the device orientation is different from the one of (X, Y, Z), like the $(X_{dev}, Y_{dev}, Z_{dev})$ frame shown in the picture. If the user now decides to start navigating forward again, all the connections between the device and the VP fields will be re-established and a jump in the vision will be noticed (from (X, Y, Z) to $(X_{dev}, Y_{dev}, Z_{dev})$). Since the 5^{th} joint is used just to establish an increment, it will not cause any jump: when changing to forward navigation, the memorized camera angle around X-axis will be incremented by a "infinitesimal" quantity depending on the angular velocity. To avoid, somehow, the cited problem, the viewpoint is made smoothly rotating around its Z-axis, till it gets to the current orientation defined by the device.

In the other configuration (section 7.2.2), a similar approach is not possible because if the tip is bent, a rotation in α (as indicated in figure 7.8) can make the tip to hit on a wall, against the user's will. If the orientation is not equal (with a certain tolerance) to the one associated to the recorded position (the same case of figure 7.12), the navigation is stopped and a force is sent to the device to prevent the user from positioning the stylus in a position with negative Y component (which would mean a forward movement). This force is proportional to the product of two terms. The first is the difference between the current device orientation about X-axis and the memorized orientation about the same axis and the second one is similar to the first, but referring to the Z-axis. This can be mathematically express as follows :

$$F_{2send} = k \cdot (\beta_{current} - \beta_{mem}) \cdot (\alpha_{current} - \beta_{mem}),$$

where F_{2send} is the force to send to the device and by *current* and *mem* is meant current angle read by the device and the memorized one, respectively (the angle α and β are those indicated in figure 7.7).

The application will indicate to the user how to move the device so as to get to the correct orientation and continue the navigation. The fundamental matrices in this implementation are two: T_{camera} and T_{base} and any of them can be calculated from the other. T_{camera} determines the observed scene and T_{base} is inescapable for the right translation of the device movement into the tip

movement.
Since the user will see what T_{camera} indicates, it is necessary to know the "history" of T_{camera} for rendering the backward movement. So the options here are two: memorizing T_{base} in the vector and then, in backward movement, calculate the correspondent T_{camera} each clock time or memorizing T_{camera} and then, when the backward movement finishes, calculating the correspondent T_{base} for continuing with the usual transformation of the base. The second one seems to be the reasonable and efficient: the memory occupied by the two options will be the same but the second solution will imply just one "hard" calculation at the end of the backward movement, instead of any clock cycle. Reminding the relation between T_{camera} and T_{base}:

$$T_{camera} = T_{base} \cdot T_{(0,-d,-\frac{l}{2})} \cdot T_{(x,-(n-1)\xi)} \cdot T_{(0,d,-\frac{l}{2})}$$

when the user changes from backward to forward the correct T_{base} to be set can be calculated as follows:

$$T_{base} = T_{camera} \cdot T^{-1}_{(0,d,-l/2)} \cdot T^{-1}_{(x,-(n-1)\xi)} \cdot T^{-1}_{(0,-d,-l/2)}.$$

In this implementation, the "pseudo-functions" *Read_ device_ value(), Update_ calibration()* and *Compute_ force* of the "Haptics_ loop()" algorithm are identical to those of the other implementation (7.2.1), whereas the central body of the algorithm changes. It can be schematized like this:

7.3 Collisions and Haptic Renderer Chosen

All the possible renderer and their characteristics were already described in section 5.3. To choose the correct Haptic Renderer, it has to be considered the kind of virtual object that has to be rendered. A bronchial tube has very thin walls and concave surfaces and so there can be some problems with a generic renderer. Using OpenHaptics renderer, for example, there had been problems of proxy falling through the surface. This can be attributed to the fact, also mentioned in [8], that when there is a collision the proxy stays on the surface but the obe can penetrate the surface itself. If and volume is thin, the probe can totally pass the volume making the control algorithm "think" that there is no collision. If no collision is detected, the proxy will be positioned in the same place as the probe and so a passing through would happen. The God Object method resolves, somehow, this problem by memorizing the "story" of the proxy: in plain terms, the method remembers which surface was first touched and so, if the probe passes on the other side of the volume the control algorithm knows that there is still a collision and keeps on exerting the reaction forces. On the other hand, GodObject renderer has some problems with acute concave intersection of surfaces [8]: if the user is pressing into one surface and sliding down, the god-object will cross to the negative side of the surface before the haptic interface will and the constraint will not be activated (fig. 7.13).

This problem can be tolerable considering two more aspects: the shape of the virtual reconstruc-

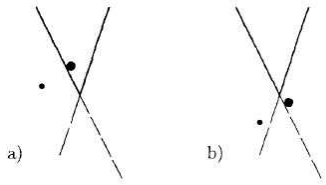


Figure 7.13: Picture taken from [8]: For an acute concave intersection of surfaces the god-object will be able to cross one of the surfaces unless special precautions are taken. The large dot represents the position of the god-object and the small dot represents the haptic interface. a) If the user is pressing into one surface and sliding down, b) the god-object will cross to the negative side of the surface before the haptic interface will and the constraint will not be activated.

tion of the bronchial tree and the kind of contact between tip and bronchial wall. With regard to the first, the triangle faces that compose the virtual bronchial tree rarely assume this kind of acute configuration. With respect to the second aspect, GodObject presents problems when the proxy is sliding on one surface, but, in this case, when the bronchoscope tip beats on a wall it is pushed back at once, so the risk of sliding on one surface and passing through another surface is very small. Furthermore, this renderer also allows custom made surface, which is fundamental for this work. For these reasons GodObject render seemed to be the most appropriate for this application.

Before choosing how to manage collisions and related forces, we have to refer to the real bronchoscopy: when the doctor is inserting the tube or hits obstacles, the greater force he perceives is along the pushing (or pulling) direction, i.e. the longitudinal tube central axis. Since it is not possible to transmit momentum, an approximation of what the doctor feels can be a force along the device Y-axis (which is the axis where the advancing is determined) every time there is a collision with the surface.

In H3D, the management of the forces exerted in a collision is assigned to a function in the surface node. So, to characterize these forces, a new H3D surface node has to be created which defines a new surface that reacts with a force along device Y-axis to any contact.

To create a custom surface in H3D, it is necessary to create a class inheriting from H3DSurfaceNode. Before doing this, a custom made class inheriting from HAPISurfaceObject has to be provided (in HAPI). In other words, the haptic behaviour is defined in HAPI and H3D transforms the HAPI new surface in a node that can be used in the X3D scene-graph.

A surface object in HAPI is an object that defines the haptic properties of a geometric shape, such as stiffness and friction. It is responsible for generating forces at a local contact point on a shape. The base class of all such objects is HAPISurfaceObject and there are several surfaces available in HAPI [5]. The surface object is responsible of two things: moving the proxy and calculating an interacting force.

When defining a new surface the virtual function `getProxyMovement(ContactInfo & ci)` in HAPISurfaceObject is used to define the proxy movement.
The user should call the function

```
ci.setLocalProxyMovement(Vec2f \& pm )
```

to set the proxy movement. The rendering system will try to move the proxy according to the specified movement but might be stopped by colliding with other shapes along the path to the new position. The proxy will then stay at this collision point instead of moving all the way to the specified position. When calculating the proxy movement and the interaction force the user has access to a ContactInfo object. This object contains many information about the contact point and the contact surface. In this case, the argument of `setLocalProxyMovement` is the `Vec2` composed by the x and z component of `ci.localProbePosition()`, which returns the position of the proxy in the contact point local coordinates[8]. This means that the proxy will move on the surface with the local x and z determined by the device values.

After the new position of the proxy has been calculated, the interaction force with the surface has to be determined. For the most common surface types this is usually a linear spring force pulling the device back towards the proxy. In this case, a force along Y-axis has to be rendered. To achieve this, it is sufficient to characterize the virtual function `getForces(ContactInfo & ci)` as follows:

```
1        void myHAPISurface::getForces(ContactInfo &contact_info){
2    double eps=0.01;
3    int forceSgn;
4    Vec3 force_to_render, VPVec;
5    Vec3 localProbe= contact_info.localProbePosition();
6    Vec3d probe_to_origin=contact_info.globalOrigin()-contact_info.
         globalProbePosition();
7    hdevSur=static_cast<HAPI::PhantomHapticsDevice*> (contact_info.hapticsDevice());
8
9        VPVec=(Matrix3f(hdevSur->getOrientation())).getColumn(2); //getting the
             third column of orn Mtrx = VP z-axis
10   //if (probe_to_origin.dotProduct(-VPVec)>0) forceSgn=1;
11   //else if (probe_to_origin.dotProduct(-VPVec)<0) forceSgn=-1;
12   //else forceSgn=0;
13   forceSgn=-1;
14   force_to_render=forceSgn*(localProbe)*20;
15   Matrix4d CalSur=hdevSur->getPositionCalibration();
16   Matrix3d CalibRotPart= CalSur.getRotationPart();
17   Matrix3d invCalibRotPart=CalibRotPart.inverse();
18
19   contact_info.setGlobalForce(Vec3(invCalibRotPart*Vec3(0,force_to_render.y,0)));
```

[8]The local coordinate system is constructed with the proxy position as the origin, the contact normal as the Y-axis and and two arbitrary perpendicular axis in the plane as X and Z-axis.

```
20 }
```

Calculating the sign in the commented way (`forceSgn`, line 10 and 11), this surface considers both cases if collision occurs when going forward and going backward (even if it may not be strictly necessary with this kind of navigation) [9]. If the collision occurs when going forward the HIP will be pushed upward and in the case that the collision occurs going backward the HIP will be pushed downward. The kind of navigation implemented make this case impossible, because the proxy will go backward traveling on the same path it made when going forward. In this work the `forceSgn` will be always positive, because it will help to go back inside the bronchial tree if a falling through occurs.

Provided that a custom made class inheriting from HAPISurfaceObject exists all that has to be done is to subclass H3DSurfaceNode. To subclass H3DSurfaceNode start by creating a constructor, fields and the database interface just as for any other node (see [12], section 4.2.1). To make this new node a functional surface node add an instance of the class created in HAPI to the variable "hapi_surface". This is usually done by overriding the initialize function of H3DSurfaceNode:

```
1 void MySurface::initialize() {
2     H3DStiffnessSurfaceNode::initialize();
3     hapi_surface.reset(
4             new HAPI::myHAPISurface( stiffness->getValue(),
5                                      damping->getValue(),
6                                      0, 0,
7                                      useRelativeValues->getValue() ) );
8   }
```

Created the new surface node, it can be used in the scene graph and set as characteristic of the bronchial tube object. Every time there will be a collision between the "virtual" bronchoscope tip and the bronchial tube, a force along device Y-axis will be sent to the device itself. The node `mySurface` will be inserted inside the *Appearance* node.
The reconstructed lungs model is a triangle face set and every face has an orientation. The node `HapticsOption` has a field called `touchableFace` which specifies which sides of the shapes to render haptically. If "BACK" only the back side of can be felt, "FRONT" only front side and "FRONT_AND_BACK" both sides[10]. To avoid falling through caused by the fact that the interior side is not the touchable (for possible errors in the model), this field is set to "FRONT_AND_BACK".

[9] This surface is defined by the class `myHAPISurface`
[10] The "BACK" and "FRONT" side are determined by the normal to the triangle surface

7.3.1 C++ implementation

The aim of this section is not to show and describe all the implementation code, but the real aim is giving the basic idea that lays under the communication between the device and the viewpoint and the setting of all the described parameters and matrices.

H3D allows its field of the same type to communicate one another. This can be done with something called *routing*. A route between field A and field B means that if something changes in field A an event message is sent to field B to let it know that A has changed and B can take appropriate actions. The values of the fields are updated using lazy evaluation. This means that the value of the field will not be updated unless some part of the code asks for its value (with e.g. the getValue() function). Fields have a member function called `update()` that takes care of updating the value. By default it just copies the value of the incoming event, but it can be changed to do any arbitrary calculation by specializing the update function. The default update function for an `SField`[11] looks something like:

```
1 class SFFloat: public Field{
2     virtual void update() {
3         value = static_cast< SFFloat>(event.ptr)->getValue();
4     }
5 }
```

where `event` is a member variable that contains a pointer to the field that caused the event and a time stamp with the time the event occur [12].

Sometimes, the existing fields could not be sufficient or the update function must be modified to achieve one's objectives. In these cases, H3D gives to the developer the possibility to build his own fields choosing the type and the number of the input parameters and the type of the returned value.

Sometimes lazy evaluation is not desiderable, it is wanted the update function to be called as soon as an event is received. This can be done by specifying the field to be an `AutoUpdate` field. In C++ this would be done as:

```
1 class PrintInt32: public AutoUpdate< SFInt32 > {
2     virtual void update() {
3             SFInt32::update();
4             cerr << value << endl;
5     }
6 }
```

which creates a field that, as soon as it receives the event, print on screen the value it received. Once constructed the desired classes with the respective update function, the field can be built by

[11] An SField is a field that contains a single value of some type and the field type is named depending on the type of the contained value.

creating an instance of this class. To make the different field to communicate one another, it is sufficient to route them in the desired order.

In the implementation of section 7.2.1 three class of this kind have been implemented. The first field (called `devNavigatorTrsf`) is "in charge" of reading the device position value and save it in a variable. Furthermore, the update function of this field modifies other two variables: the linear velocity depending on the y device position and the blocking of the device movement along its X-axis. This blocking is made by sending a force proportional to the x distance from the origin. The event routed to this field will be the position read from the device. This means that every time the device position changes, the update function of this class will be called and all the values will be, exactly, updated.

The second field is `devNavigatorOrn`. This memorizes the orientation of the device in a variable and determines the camera angular velocity (about its X-axis). The event routed to it is the device orientation.

Finally, the class `viewUpdater` that contains all the calculations of the transformation matrices and that takes care of setting the viewpoint and calibration matrices to the right value. The update function of this field is the one that moves everything and allow the navigation, so it is wanted this function to updating continually. For this reason, the event routed to this field is the `fraction_changed` of the clock: every clock time the update function will be called. The values of linear and angular velocity and device position and orientation provided from the other two fields need to be global variable, so as to be utilized in the viewUpdater class to execute all the calculations.

On the other hand, in the implementation described in section 7.2.2 the necessary classes are only two. The class that, in the other case, managed the angular velocity is no longer needed, since the angle about the camera X-axis correspond to the device orientation about the X-axis as explained in the corresponding section (in this implementation the angle ξ of figure 7.7 is not modified by angular velocity). Excepting for the `devNavigatorOrn`, the other two classes are in this implementation too and have the same role. Obviously, the `viewUpdater` will contain the operations (setting VP and calibration matrices) of the specific described implementation.

Referring to the pseudo-code of section 7.2, there is a correspondence between that code and these classes. `devNavigatorOrn` and `devNavigatorTrsf` represents the `Read_device_info()` function and the `viewUpdater` is in charge of executing `Compute_control()`, `Update_calibration` and `Update_viewpoint()`. As already explained, the `Compute_forces()` function is managed directly by the surface.

As mentioned, to route the different fields the corresponding class objects have to be created. In this case, it was done as follows:
First creating the new needed fields

```
1 // Read_device_info (position and orientation)
2 devNavigatorTrsf *devNavigatorTrsfObj = new devNavigatorTrsf();
3 devNavigatorOrn *devNavigatorOrnObj = new devNavigatorOrn(); // just in ``
      controlling directly the camera'' implementation
4 // Update_viewpoint(), Update_calibration() and Compute_controls()
5 viewUpdater *viewUpdaterObj = new viewUpdater();
```

and then, when loading the scene, they have to be connected with the desired field:

```
1         // reading the hdev values
2     hdev->devicePosition->route(devNavigatorTrsfObj);
3     hdev->deviceOrientation->route(devNavigatorOrnObj); // just in ``controlling
          directly the camera'' implementation
4     hdev->followViewpoint->setValue(false);
5
6     // starting the clock and routing to viewUpdater field
7     AutoRef< TimeSensor > ts( new TimeSensor );
8     ts->startTime->setValue( TimeStamp()+1);
9     ts->fraction_changed->route(viewUpdaterObj);
10    ts->loop->setValue(true);
```

In this code, '`hdev`' is the name of the pointer that refers to the connected device. The field `followViewpoint` (true if the device should follow the viewpoint) must be set to false. If its value is true, this settings will act on the calibration matrix so as to maintain the probe position and orientation on the screen even when the VP is modified. This setting would obviously collide with the desired behaviour, which wants the device to follow the device.

After having established the different routings, any time a device value of position or orientation changes, it will be revealed and the value will be passed to the routed field which will handle the event by changing the settings in the appropriate way.

Chapter 8

Graphics Interface

The **objective** of this module is to provide a Graphic User Interface (GUI) made using Qt, which allows the user to explore the bronchial tree with the haptic device. It must also provide an external view of the same bronchial tree, where a point will indicate the position of the virtual camera during the exploration. This GUI must have also tools to make the navigation easy and intuitive.

This section presumes to describe just the main concepts which are at the basis of the implementation, the entire code of C++ classes and the detailed functioning of the application can be found in the appendix and the accompanying CD.

8.1 Qt - H3D integration: The *QTWindow* Class

H3D provides users with an H3DViewer which is totally built using the *wxWidgets* library. Since the aim of this project is to create an application ("Lung Viewer") in Qt and nothing was available with Qt, the first step was to implement the integration of an H3D scene in a Qt window, guaranteeing keyboard and mouse interaction with the specified scene.

In H3D, the base class for all windows nodes is `H3DWindowNode`. It handles creation of windows and window properties for looking into a Scene. To implement a new window class the following virtual functions have to be specified: swapBuffers(), initWindow(), initWindowHandler(), setFullscreen(bool fullscreen) makeWindowActive(), setCursorType(const std::string & cursor_mode) and getCursorForMode(). These functions define the fundamental actions associated to any window, they are functions to create the window and set of the properties, to set whether the window should be fullscreen or not or to make the window active. The same class also has a set of functions that manage the signals coming from mouse and keyboard and allow to interact with the scene. The missing "plug" is a event handler, that can recognize the signals

Qt	H3DWindowNode
mousePressEvent(QMouseEvent *event)	virtual void onMouseButtonAction(int button, int state);
mouseReleaseEvent(QMouseEvent *event)	virtual void onMouseButtonAction(int button, int state);
keyPressEvent(QKeyEvent *event)	virtual void onKeyDown(int key, bool special);
keyReleaseEvent(QKeyEvent *event)	virtual void onKeyUp(int key, bool special);
mouseMoveEvent(QMouseEvent *event)	virtual void onMouseMotionAction(int x, int y);
wheelEvent(QWheelEvent *event)	virtual void onMouseWheelAction(int direction);

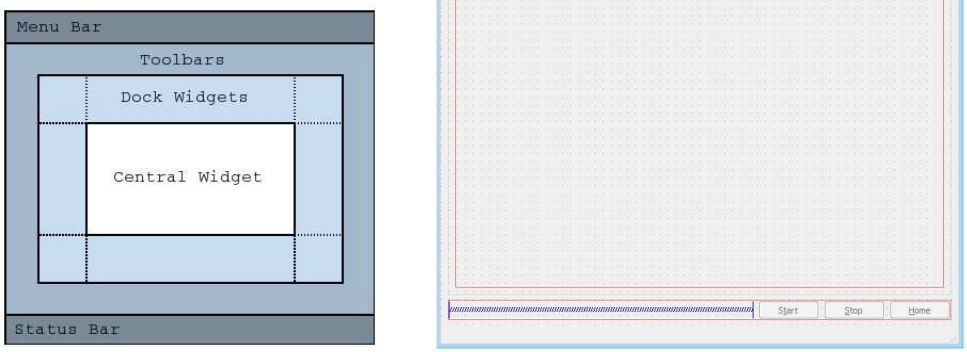
Table 8.1: Correspondence between the Qt function for catching mouse and keyboard events (left column) and the H3DWindowNode functions to manage them (right column). Each Qt function calls its corresponding H3D function, passing the right parameter.

from external devices. This event handler is provided in Qt by the functions: mousePressEvent(QMouseEvent *event), mouseReleaseEvent(QMouseEvent *event), keyPressEvent(QKeyEvent *event), keyReleaseEvent(QKeyEvent *event), mouseMoveEvent(QMouseEvent *event), wheelEvent(QWheelEvent *event). The "event" parameter includes information about which element (of keyboard or mouse) has provoked the event. These functions are just used to catch the event and call the H3DWindowNode function that handles the event caught. In tab. 8.1 can be seen the correspondence between Qt and H3D functions for acting on windows.

On the Qt side, the base class of all user interface objects is *QWidget*. The widget is the "atom" of the user interface: it receives mouse, keyboard and other events from the window system and paints a representation of itself on the screen. To achieve the Qt-H3D connection, the idea proposed in this project is to create an object which inherits from H3DWindowNode and from QWidget, so as to be able to catch the events and call the corresponding H3D function with the right parameters. The combination of this functions will allow to explore and look into the scene. To associate the window to the node `Scene`, QTWindow must be a H3D node itself. To do this it is sufficient to add the following code in the *.cpp* file of the QTWindow class, defining the name of the new node and adding it to the H3D database.

```
1 // Add this node to the H3DNodeDatabase system.
2 H3DNodeDatabase QTWindow::database( "QTWindow",
3                                     &(newInstance<QTWindow>),
4                                     typeid( QTWindow ),
5                                     &(H3DWindowNode::database) );
```

So, when a window is to be added to a scene, it will be sufficient to create the window and set the value of the field `scene->window` with the window just created.



(a) *QMainWindow layout structure.*

(b) *QtDesigner form for the LungViewerWindow, which is the application main window.*

Figure 8.1: QMainWindow layout and its relative implementation in the main application GUI.

8.2 Main Application

8.2.1 Graphic design

As already explained in previous sections, Qt supplies the user with a useful tool called QtDesigner. This helps building the main structure of the window by dragging and dropping basic widget forms (such as buttons, tags, sliders, menus...) in the empty window model and disposing them in the intended layout. Once the window structure is completed, this software allows to set properties and how the events generated by the user have to be managed. Another important feature of this tool is that it is perfectly integrated in QtCreator (the Qt ID environment): the user can create a *Qt Designer Form Class* and the *ui*[1] file and the relative class are automatically generated. Furthermore, the C++ code describing the window structure can be easily obtained using CMake tools: `QT4_WRAP_UI` reads the *formName.ui* files (an example can be the one in figure 8.1(b)) and creates the file `ui_formName.h`. This file only contains the C++ code corresponding to the form built in QtDesigner, including the connection of signals and slots (which will be explained below). In the class associated to the form, the creator and all the needed signals and slots are defined and implemented.

The application main window is a *QMainWindow*, a Qt class that provides (with classes associated to it) a framework for building an application's user interface. QMainWindow has its own layout to which you can add QToolBars, QDockWidgets, a QMenuBar, and a QStatusBar. The layout has a center area that can be occupied by any kind of widget (8.1(a)). As told before, a

[1] *ui* is the extension of the files created in QtDesigner.

QtDesigner form needs to be processed by QT_WRAP_UI. An example of the created code can be the following *.h* file and and its structure is similar for every form. Here it will be shown the code relative to the form of figure 8.1(b):

- In the first part there are all the classes of the basic widget needed to build the window:

```
#ifndef UI_LUNGVIEWERWINDOW_H
#define UI_LUNGVIEWERWINDOW_H


#include <QtCore/QVariant>
#include <QtGui/QAction>
#include <QtGui/QApplication>
#include <QtGui/QButtonGroup>
#include <QtGui/QHBoxLayout>
#include <QtGui/QHeaderView>
#include <QtGui/QMainWindow>
#include <QtGui/QMenu>
#include <QtGui/QMenuBar>
#include <QtGui/QPushButton>
#include <QtGui/QSpacerItem>
#include <QtGui/QStatusBar>
#include <QtGui/QVBoxLayout>
#include <QtGui/QWidget>

QT_BEGIN_NAMESPACE
```

- then the class and all the objects used in the window are defined:

```
class Ui_LungViewerWindow
{
public:
    QAction *actionOpen_Lung;
    QAction *actionOpen_Path;
    QAction *actionMain_Settings;
    QWidget *centralwidget;
    QVBoxLayout *verticalLayout;
    QWidget *LunPlcae_widget;
    QVBoxLayout *verticalLayout_2;
    QHBoxLayout *LungPlace_Layout;
    QHBoxLayout *horizontalLayout;
    QSpacerItem *horizontalSpacer;
    QPushButton *startButton;
    QPushButton *stopButton;
    QPushButton *homeButton;
    QMenuBar *menubar;
    QMenu *menuFile;
    QMenu *menuOpen;
```

```
20      QMenu *menuSettings;
21      QStatusBar *statusbar;
```

- subsequently all the elements composing the window are named and positioned in the layout. At the end of the setupUi function, the connection between signals and slots are defined.

```
1       void setupUi(QMainWindow *LungViewerWindow)
2       {
3               // defining, naming and positioning all the elements
4
5           if (LungViewerWindow->objectName().isEmpty())
6               LungViewerWindow->setObjectName(QString::fromUtf8("LungViewerWindow
                    "));
7           LungViewerWindow->resize(639, 526);
8
9                   // [ . . . ]
10
11          //Defining the connection between signals and slots.
12
13          QObject::connect(actionOpen_Lung, SIGNAL(triggered()), LungViewerWindow
                , SLOT(openLung()));
14          QObject::connect(actionMain_Settings, SIGNAL(triggered()),
                LungViewerWindow, SLOT(openMainSett()));
15          QObject::connect(startButton, SIGNAL(clicked()), LungViewerWindow, SLOT
                (routeAll()));
16          QObject::connect(stopButton, SIGNAL(clicked()), LungViewerWindow, SLOT(
                unRouteAll()));
17          QObject::connect(homeButton, SIGNAL(clicked()), LungViewerWindow, SLOT(
                getHome()));
18
19          QMetaObject::connectSlotsByName(LungViewerWindow);
20      } // setupUi
21
22  };
```

- Finally, the class associated to the form is defined (LungViewerWindow) and force to inherit from the *ui* class.

```
1  namespace Ui {
2      class LungViewerWindow: public Ui_LungViewerWindow {};
3  } // namespace Ui
4
5  QT_END_NAMESPACE
6
7  #endif // UI_LUNGVIEWERWINDOW_H
```

8.2.2 Functioning

As already mentioned, the *signals and slots* mechanism is fundamental for Qt programming. It enables the application programmer to bind objects together without the objects knowing anything about each other. So, by this mechanism, the widgets (and all the structure formed by that) can communicate one another or with the main process. When an event occurs (such as an *OK* button is pressed) the involved element sends a signal according to the occurred event. This signal is redirected to the bind object (of the same or another window) which will react with the slot associated to that signal. One object can throw different signals and each signal can be connected to many slots or, vice versa, many signals can be connected to the same slot. These connections can easily be defined in QtDesigner itself, by linking the different widgets and choosing the existing signals/slots or creating some new. As shown above, the connection astatement looks like this:

```
1        connect(sender, SIGNAL(signal),receiver, SLOT(slot));
```

where `sender` and `receiver` are pointers to QObject and where `signal` and `slot` are function signature without parameter names [14].

The only thing to be respected is the coherence between the parameters of the linked signal and slot: the signal and the slot must have the same parameter types and in the same order. Signals and slots are here used just with widgets, but the mechanism is implemented in `QObject` and is not limited to GUI programming. It can be used by any `QObject` class. Slots and signals have to be defined and implemented in the class associated to the form (in this case it is `LungViewerWindow`). A little inconvenience of this method is that Qt uses a preprocess MOC to translate signals and slots and so there is no access to "pure" C++ code.

Another two features of this main window deserve to be mentioned: the *LungViewerWindow* class constructor and the "load function".

The **constructor** has the following form:

```
1 /// Constructor
2 LungViewerWindow::LungViewerWindow(QWidget *parent) :
3     QMainWindow(parent),
4     ui(new Ui::LungViewerWindow)
5 {
6
7     ui->setupUi(this);
8
9     scene.reset( new Scene );
10    // create a window to display
11    QTWindow *glwindow = new QTWindow;
12    scene->window->push_back(glwindow);
13
```

```
14      ui->LungPlace_Layout->addWidget(glwindow);
15      scene->sceneRoot->setValue(NULL);
16
17      settings=0;
18  }
```

where it can be observed the assignment of the form to the window (line 7) and the creation of the QTWindow which is associated to the Scene `scene` (lines 11 and 12). At the beginning the scene root is `NULL` (no file is shown), so, when opening this main window, a black rectangle (which represent the "NULL scene") will be shown in the QTWindow position. The scene is "filled up" when the load function is called.

The other important element is the **loadFile** function. Its input parameter is the path to the file of the lung image. What this function does is to load a basic empty scene with all the general settings and fill in the two empty groups **"LUNGS"** and **"ROUTES"** defined to contain the bronchial tree input parameter and the possible guidance path, respectively. This basic file is called *base.x3d* and looks like this:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <X3D profile="H3DAPI" version="2.0">
3   <Group DEF="BASE">
4
5     <DeviceInfo>
6       <PhantomDevice positionCalibration="10 0 0 0
7                                           0 10 0 0
8                                           0 0 10 0
9                                           0 0 0 1">
10        <GodObjectRenderer/>
11        <Shape containerField="stylus">
12          <Appearance>
13            <Material />
14          </Appearance>
15          <Sphere radius="0.01" />
16        </Shape>
17      </PhantomDevice>
18    </DeviceInfo>
19    <IMPORT inlineDEF="H3D_EXPORTS" exportedDEF="HDEV" AS="HDEV" />
20
21    <GlobalSettings>
22        <HapticsOptions DEF="HaOp"/>
23        <DefaultAppearance>
24            <Appearance containerField="defaultAppearance">
25                <SmoothSurface stiffness="0.5" />
26            </Appearance>
27        </DefaultAppearance>
28    </GlobalSettings>
```

```
29
30      <Viewpoint DEF="VP" position="0 0.0 3" />
31      <!-- Disable the headlight since we want an overhead light -->
32      <!--NavigationInfo headlight="FALSE" type="NONE" /-->
33      <!--Position a light overhead, slightly off-centered -->
34      <!--DirectionalLight direction="0.1 -1 0.1" /-->
35      <!--DirectionalLight direction="0.1 1 0.1" /-->
36
37
38      <!--LUNGS-->
39      <Group DEF="LUNGS">
40
41      </Group>
42
43
44      <!--X Y Z AXIS-->
45      <Group DEF="ROUTES">
46      </Group>
47
48    </Group>
49 </X3D>
```

where:

- `DeviceInfo` node defines the device settings such as the renderer used and the shape of the proxy (a sphere in that case).
- `IMPORT` node detects the device and assigns it a name that will be used to refer to it.
- `GlobalSettings` establish some global haptics options, such as the touchable face or the stiffness of all the surfaces.

An X3D file defines a scene-graph using an XML-like syntax characterized by nodes and tags. To each node a string can be associated. This string (inserted with the `DEF` command) gives a name to the node, which can be used for further references. H3D has a class named `DEFNodes` which provides a mapping between defined DEF names in X3D and the nodes they refer to. The insertion of the input file inside `base.x3d` is possible thanks to this tags on the nodes. When an X3D file is loaded, all its defined nodes can be collected in a `DEFNodes` object and can be used to modify the graph from the C++ environment. How to insert "lungs scene-graph" is shown and commented below. After having created the DEFNodes object `myDefNodes1` and referred to the root node (`Group base`) of the `base.x3d` file, the group which will correspond to the LUNGS node is created

```
1 Group* base= new Group();
```

```
2 myDefNodes1->getNode("BASE",base);
3 // creating the group which will correspond to the LUNGS node
4 Group *lungs= new Group();
5 myDefNodes1->getNode("LUNGS",lungs);
```

Then, the node that receives the input file graph must be created (called `lungsNode`). The command createX3DFromURL create H3D nodes given X3D data as a URL and `lungsNode` will be pointing to the first node of the X3D scene-graph contained in `fileName`.

```
1 // inserting the scene graph defined by fileName as children of the ''lungs'' group
2 X3D::DEFNodes *lungsDefNodes= new X3D::DEFNodes();
3 AutoRef<Node> *lungsNode;
4 lungsNode=new AutoRef<Node>(X3D::createX3DNodeFromURL( fileName.toStdString(),
      lungsDefNodes));
```

The last step is to insert the graph into the existing file and then set this complete scene graph as the sceneRoot:

```
1 if (t_lungs) t_lungs->children->push_back(lungsNode->get());
2 if (lungs) lungs->children->push_back(t_lungs);
3
4 //[...]
5
6 scene->sceneRoot->setValue(base);
```

The load function (called `loadPath`) for adding the calculated path in the scene is based, more or less, on the same idea. A node containing the path is created by the function `createX3DNodeFromURL` and this node is inserted as children of the group `base`, which is declared as private variable of the `LungViewerWindow` class.

In the implementation of controlling the camera from the base of the tip, an additional tool is needed. As explained in chapter 7, in this implementation, it is not possible to pass from backward to forward if the current device orientation does not correspond to the one associated to the "old" memorized position. With the aim to help the user to position the device in the right orientation, a little window has been created. This window (fig. 8.2) indicates to the user by red arrows in which direction he/she has to rotate the stylus or if he has to move its tip up or down.

This window is obtained using the QPainter class, which performs low-level painting on widgets and other paint devices (see [13] and [14] for more details).

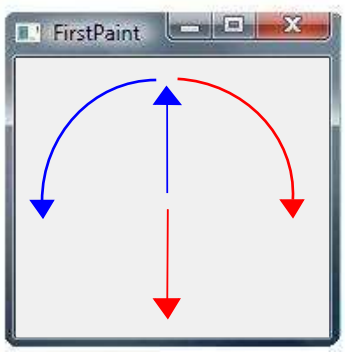


Figure 8.2: Window indicating the movement to make with the device to align the device orientation with the memorized one.

8.3 Two different views

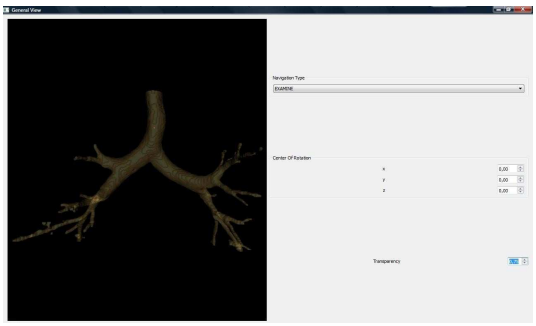
As said in the introdcution of this chapter, the other main feature of the application is having two different views: one showing the scene seen by the bronchoscope camera and the other showing the external view of the lungs with a sphere indicating the position of the tip inside the bronchial tree.

In the attempt of creating two windows (QTWindow) containing the same scene but with different viewpoint, H3D presents some problems: when the user acts on the image with the fixed viewpoint this stays still whereas the other scene reacts as if this last scene was receiving the event. The internal H3D navigation system is not built to handle two viewpoints at the same time and navigation in both windows. Even consulting the H3D Forum, it does not seem to be possible or easily implementable.

In this project the total independence between scenes is achieved working with separate processes: one dealing with haptics navigation and the other with the external view. This way of implementing also assures navigation for both windows, even with different viewpoints. However, a new problem arises, concerning the fact that the haptics device can communicate with just one application at a time. So, it has to be found a method to make the two processes communicate one another so as also the second one can know the tip position and orientation and, therefore, positions the indication point in the right coordinates inside the bronchial tree.
The solution adopted here is to create a shared memory where the both processes can access, read or write data. To achieve that, BOOST library is used. This library provides, among other packages, the *Boost.Interprocess* which allows to create and manage a shared memory and, therefore, a communication between processes in the same computer (or machine in general).

The *Lung Viewer* main window is the window that receives the haptics device information and it will be the main thread. This application is also responsible of updating the calibration field,

(a) *Lung Viewer window which visualizes the bronchoscope camera view.*



(b) *Window used for the general view. The figure is more transparent and a little red point can be seen at the mouth of the trachea.*

Figure 8.3: The Complete Graphic User Interface

apart from the communication with the device. The main process creates the shared memory segment in which a vector of matrices will be saved. This vector contains the matrices decribing the "historical" positions and orientations of the bronchoscope tip. They are 4×4 matrices expressing the transformation between the world coordinates and the tip coordinates. Every updating cycle, the main process adds to the shared vector the last matrix value. The second process just reads from the shared memory segment the last vector value, which represents the current position. It is "in charge" of showing the general view of the lungs and translate a little sphere according to this read value. Every time a new file is opened, the shared memory is cleared to free the space and avoid that the values regarding different navigation will affect the current one. Finally, being two parallel processes, it is made possible to interact with the scene rendered in general view window by mouse or keyboard, without affecting the other one.

Chapter 9

Environmental Analysis

Since this work is purely a software development, it does not affect the environment directly.

More or less the same approach was used to develop the different modules. The first phase, always, consisted of consulting books, articles and manuals and doing some tests to know the tools which will be used in the project. This implied the use of paper (new, used or recycled) to print some essential part of documents or write notes or calculations and the use of computer to make tests or also read documents. The following phase was more centered on the development of code and tests using the computer and the device, when needed. In this phases the use of paper may decrease (just use to print some important code) but, on the other hand, the consumption of electric energy increases. Since this software should be distributed and tested in two hospitals, at least two CD must be burned and distributed. The effects on the environment can be summarized as follow:

- Emissions in the atmosphere caused by a continuing use of a computer. Considering an average power consumption of 360 W for 6 months (24 day per month) and a average time of 10 hours per day, this consumption can be estimated in about 520 kWh.
- Emissions caused for the use of the haptics device. Considering an average power consumption of 40W and an average use of 2 hours per day in only the second part of the work (4 months), the electricity consumption can be estimated in 192 kWh.
- Paper and ink consumption to print documents in general or part of interesting code and to write notes or calculations. The paper used was recycled, written on one page or new sheets.
- CDs for the distribution of the software (at least two for the two hospital which are collaborating in this project) that need to be recycled when not used anymore.
- Emissions as consequence of the transport used to go to the laboratory or libraries.

A last consideration to make is that all the paper used in the project and considered useless at the end of the project itself has been recycled.

Chapter 10

Costs Analysis

The costs deriving from this work can be divided into two groups: costs related to the physical equipment or material and those related to the staff work (composed by doctor engineer, a PhD student and a new engineer.)

Physical Equipment

Equipment / Material	Unit Price [€]	Total Cost [€]
Phantom Omni	1780	1780
PC Dell T1500	1000	1000
Fungible Material (CD, Sheets of paper...)	-	20
Total	-	3000

The indicated price may seem a high price for a tower PC, but a good graphical board with high calculation capacity is needed to process the "heavy" images of bronchial tubes. It need also a Fire-Wire (*IEEE 1394*) to communicate with the haptics device.

Working Staff

Profession	Hours	Professional Fee [€/h]	Total Cost [€]
Senior Engineer	100	100	10000
PhD Student	100	80	8000
Engineer Junior	1080	35	37800
Total			55800

The cost of transport to get to the laboratory or to get where the meeting took place are included in the professional fee.

The estimate of this project can be around 58.800 €, which can be an hypothetical budget to carry out this work.

Chapter 11

Results

A software has been implemented to allow the navigation in the VB by means of a haptic device, satisfying the requirements of the application expounded in section 4.

The software allows the user to observe the virtual reconstruction of the bronchial tree of a specific patient and to explore it by a haptics device. To achieve that, the bronchoscope was modeled and the transformation **matrices** between device and virtual world were calculated **considering the model**.

The movements made for the **virtual exploration** are very similar to the ones made in real bronchoscopy and two implementations are proposed, which have to be evaluated by competent medical staff.

The **collisions** are managed in the right way avoiding the proxy from exiting from the bronchial tube and rendering a force in device Y-axis. This makes the stylus lifting and, consequently, causes a backward movement of the camera that allows the observer to see his way and go on with the navigation. The **external view** helps the user's bearings showing the position of the camera inside the bronchial tube.

The specification regarding the libraries to be used have been satisfied too: **graphics and haptics rendering** has been developed with H3DAPI and HAPI and the **Graphic User Interface** by Qt. Defining the project by a **CMake** file enables to build the project both for Linux and Windows.

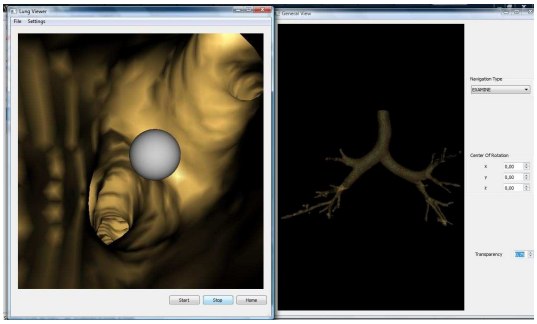
In the tests made so far, the software does not present any strange behaviour. It has a slow start up depending on the size of the `.wrl` loaded for the navigation. The haptic renderer provides good results but in some cases does not totally avoid falling through problems. It depends on the quality of the reconstruction and on the shape of the hit zone. If a falling through may occur, the provided going backward movement, assure the return of the camera inside the bronchus and the

user can start again his/her navigation. The interface will be composed of two separate windows and each of them is associated to one process (navigation or external view). The user can manage separately the two windows, set their dimensions and move them independently or choosing which one to show on the screen. Moreover, it is guaranteed the total independence between the two scenes for the user, that can modify one without the other being affected.

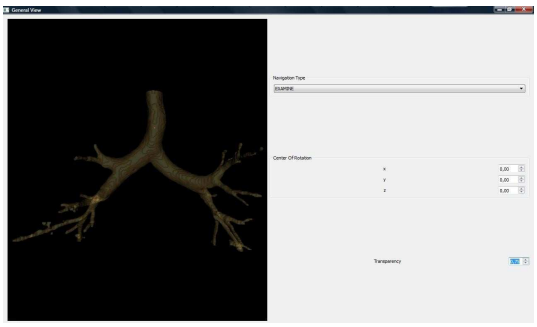
The forces sent to the device (caused by collisions or to constrain it to a position with a null X-component) make the device vibrating sometimes. The reason of that could be related with several issues. As described in [7] haptic interfaces can only exert force with limited magnitude and not equally well in all directions, thus rendering algorithms must ensure that no output components saturate, as this would lead to erroneous or discontinuous application of forces to the user. Another issue could be that haptic-rendering algorithms operate in discrete time whereas users operate in continuous time: while touching a virtual object, the virtual sampled probe will always lag behind the probe's actual continuous-time position. Due to this beahaviour, the user needs to perform less work than in reality, when pressing on a virtual object. The problems can come when the user releases: the virtual object returns more work than its real-world counterpart would have returned. This "over-reaction" can cause an unstable response. Finally, haptic device position sensors have finite resolution. Consequently, there are always quantization errors, attempting to determine where and when contact occurs or just the device position (needed, then, to calculate velocity or determine whether or not forces has to be sent). Although users might not easily perceive this error, it can create stability problems.

Figure 11.1 shows some screen-shots of the created application. Moreover, some videos have been made to show better and attached in the accompanying CD.

(a)

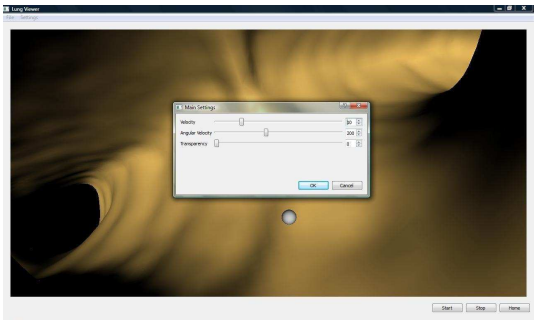


(b)

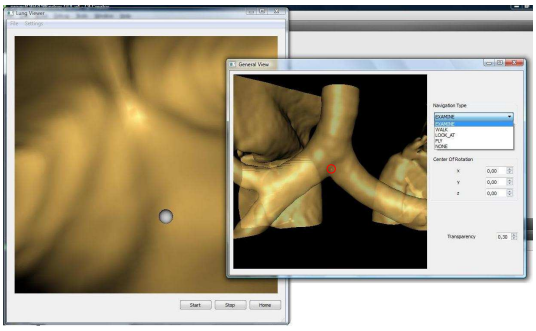


(c)

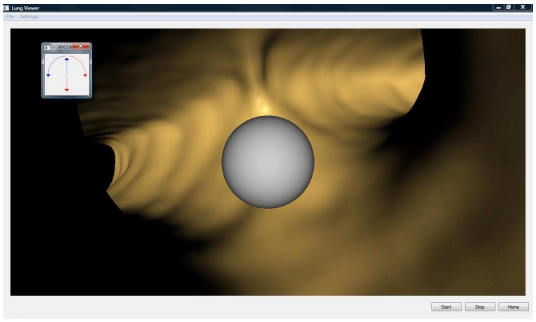
(d)



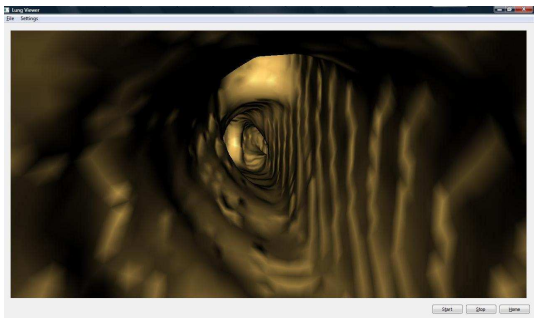
(e)



(f)



(g)



(h)

Figure 11.1: Screenshots from the application

Chapter 12

Conclusions

All the objectives of the beginning have been attained with good results. This work can be considered also a sort of experiment to test new libraries and merge them to "blend" cleverly the quality of each of them.

Qt turns out to be a very good tool for developing GUI and widgets in general. It enables the user to accomplish the GUI very quickly, using all its tools, which are perfectly integrated in the QtCreator IDE. The Qt Reference Documentation is very complete and clear and plenty of useful examples.
H3DAPI integrates haptic and graphic rendering in a very good way. It represents a powerful development tool that consents to the software developer to save time and achieve very good results. `HAPI` allows a total control of the device and of the haptic side of the scene. H3DAPI build nodes from HAPI structures, which can be used in the scene-graph making all the development easier and the structure clearer. On the other hand, when creating something new regarding haptic rendering, it is needed the HAPI structure first and then the node with H3DAPI can be defined calling the HAPI object just created.

Finally, another positive consideration can be done about *Python*[1], even though it was never mentioned before. It cannot be numbered among the fundamental software used developing this work, but it has turn out to be a good tool for testing operation. It is a very easy and intuitive programming language, which H3D uses for many examples and to implement useful tools. Since it is an interpreted language, its efficiency may not be as good as C++ written programs, so the choice to develop everything in C++.

Despite being a basic version, it can be defined a good tool for navigation. It allows also to

[1] http://www.python.org/

adjust some important parameters like transparency, navigation type and velocity or the position of the center of rotation, offering more possibilities to carry out the navigation and analyze more specifically some parts. It can be used to carry out the virtual bronchoscopy more quickly and in a more intuitive way. It acquires a "value-added" inside the bigger project already mentioned at the beginning: it is the application which make it possible to integrate the results of both virtual reconstruction from CT images and path-finding modules.

Chapter 13

Future Work

Using this work as basis, there are many possibilities to add features and expand the field of application. Some of them have already been considered as natural continuation of this project.

13.1 "Improving the Feeling"

Two aspects can be added to make the navigation more real. One is to make the virtual broncoscope tip to turn if one side of the tip is touching a bronchus wall, so as to better render the real bronchoscope turning action. In fact, what really happens in real bronchoscopy is that the doctor (when getting to a fork) turns the tip in the desired direction and push the tube forward; the bent bronchoscope tip hits on its lateral side against the bronchus wall and, then, keeping on pushing the bronchoscope, it slides on the bronchus wall and actually turns as desired. To better render the feeling of this behaviour and of bronchoscopy in general, a solution could be to give a different interacting shape to the proxy. In this case the haptic interaction point will be a bronchoscope tip model. This would be complicated to calculate the reaction force from the collision points of the virtual tip and to translate them in a meaningful way to the device (i.e. send a force to the device that render, somehow, the real feeling).

13.2 Guidance

This application counts with the insertion of a planned path that indicates to the doctor the route he/she has to follow to reach the target point (for example a cancer nodule in a lung peripheral zone). Haptics properties can be exploited to lead the doctor exploration following the proposed route, by sending forces to the device that indicate the doctor how to actuate.

Another thing that can be useful to add is the vision of the DICOM images which the doctors are very familiar with. So, during navigation, they can see the 3-dimensional reconstruction and identify where they are also in the two planes (sagittal and trasversal) in DICOM images.

Appendix A

User Manual

When the *Lung Viewer* is opened, appears as seen in figure A.1. To start the navigation is necessary to load a scene. This can be done by clicking on the *File* menu and choosing *Open->Open Lung*. Once selected the file to open, the scene will appear on the main window (named Lung Viewer A.2(a)) and a second window will be open showing the general view of the lungs and a little spot indicating the position within the bronchial tube during the navigation (fig. A.2(b)). To insert the planned path is sufficient to click on the other item of the File menu and choose the desired file.

Lung Viewer

Lung Viewer presents 3 buttons:

- **Start** allows the user to start the navigation by the haptics device.
- **Stop** interrupts the navigation and the camera stays at the same position and orientation as when the button is pressed.
- **Home** is like a reset button: it brings the situation as it is at the beginning.

These buttons are very useful if the user loose his/her way and wants to start from the beginning or to stop the navigation if some problem occurs.

It has also the menu *"Settings"* (fig. A.3)that open a dialog window where the user can change some parameters:

- **Velocity** allows the user to change the linear forward velocity. The biggest is the value the less the user has to move the device to obtain a certain displacement velocity.

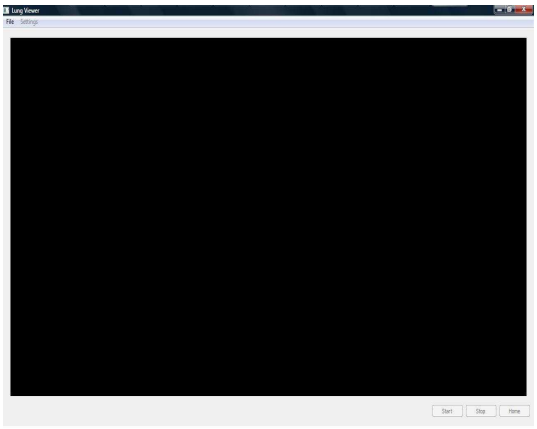
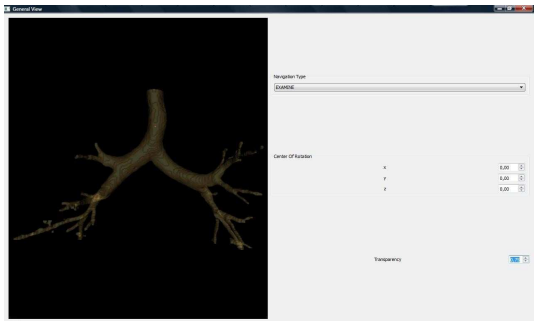


Figure A.1: The appearance of the main window when the process start.

(a) *Lung Viewer window which visualized the bronchoscope camera view.*



(b) *Window used for the general view. The figure is more transparent and a little red point can be seen at the mouth of the trachea.*

Figure A.2: The Complete Graphic User Interface

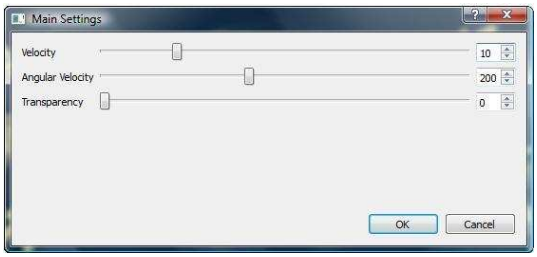


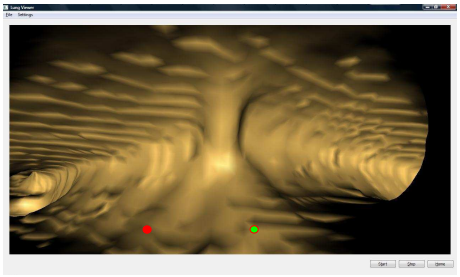
Figure A.3: The Main Settings dialog window.

- **Angular Velocity** available only in the navigation controlling directly the camera. It change the velocity of increasing-decreasing the tip angle.
- **Transparency**, as the word tells, is used to set the transparency of the bronchial tube: its range goes from 0 to 1 where 1 is totally transparent and 0 is the natural color.

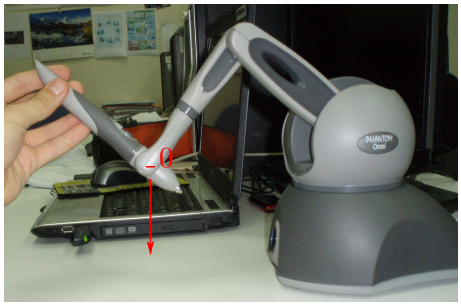
General View

This second window opens only when a new file is opened in Lung Viewer. It presents three Group Box: one showing the current navigation type and letting the user to modify it, another one showing the coordinates of the center of rotation and the last one indicating the transparency.
The center of rotation is the point which the bronchial tube will move around when the user will act on the window with the mouse. Furthermore, it is also the point where the camera comes nearer when zooming by mouse. To change the orientation of the viewpoint is only necessary to click on the image and translate the mouse in the desired direction.
The navigation type determines the user interface capabilities of the browser. There are many type of navigation [15]:

- WALK: is used for exploring a virtual world on foot or in a vehicle that rests on or hovers above the ground. It is strongly recommended that WALK navigation define the up vector in the $+Y$ direction and provide some form of terrain following and gravity in order to produce a walking or driving experience.
- FLY: is similar to WALK except that terrain following and gravity may be disabled or ignored. If the type is "FLY", the browser shall strictly support collision detection.
- `LOOK AT`: is used to explore a scene by navigating to a particular object. Selecting an object with LOOKAT, it moves the viewpoint directly to some convenient viewing distance from the bounding box center of the selected object, with the viewpoint orientation set to aim the view at the approximate "center" of the object and it sets the center of rotation in the currently bound Viewpoint node to the approximate "center" of the selected object.



(a) *Example of a scene showing a bronchial tree during the forward movement*



(b) *To obtain a straight forward movement, the user has to bring the HIP in a negative position about y component. The arrow here is not indicating any references coordinate, it indicates just the suggested movement to perform.*

Figure A.4: How obtain the forward movement and the correspondent result.

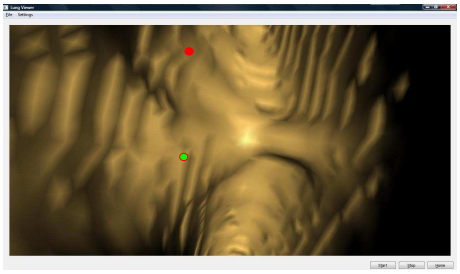
- `EXAMINE` navigation is used for viewing individual objects. EXAMINE shall provide the ability to orbit or spin the user's eyepoint about the center of rotation in response to user actions. The center of rotation for moving the viewpoint around the object and determining the viewpoint orientation is specified in the currently bound Viewpoint node.

- `NONE` navigation disables and removes all browser-specific navigation user interface forcing the user to navigate using only mechanisms provided in the scene, such as Anchor nodes or scripts that include loadURL(). NONE has an effect only when it is the first supported navigation type.

A.1 Example of navigation

In this section it will be shown in practice the movement the user must do to explore the scene and turn into a bronchus. The forward movement is obtained by pushing the device in a position with negative y component (figure A.4(b)). When the observer will get to a fork of the bronchial tree (figure A.4(a)), he will need to turn into the desired bronchus and proceed with the navigation. Considering how the bronchoscope is modeled in this work, its tip is able to turn only on the vertical plane (Y, Z) with respect to the camera frame, in other words just up-down movement is consented. To enter in bronchi "lying " on other planes, a combination of more movements is needed. In this section, it will be analyzed how to turn into a left bronchus, considering the trachea and the two main bronchi in a horizontal position as shown in figure A.4(a).

To turn left, the user must position the plane which the tip moves on in the position needed

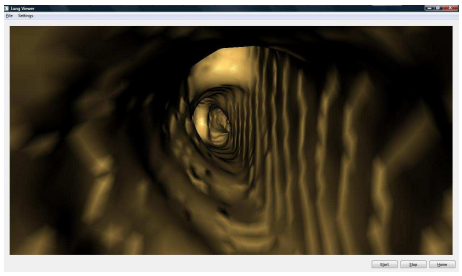


(a) *The rotation of the camera about its Z-axis. The camera is rotated counterclockwise, so the image appears rotated clockwise. This can be seen comparing the spots with those of figure A.4(a).*

(b) *To obtain the result shown in the picture A.5(a) the 6^{th} joint of the device has to be rotated counterclockwise as shown.*

Figure A.5: The rotation around camera Z-axis.

to turn in the desired direction. In this case he can turn counterclockwise the camera (acting on joint 6 in the same direction, see fig. A.5(b)) so as to position the tip in a way it can turn in the wanted bronchus. After this rotation, the scene observed will be that on of figure A.5(a) where the bronchi, now, are positioned in the vertical plane. Comparing figure A.4(a) and A.5(a) can be seen that the left bronchus now become the upper one. At this point, the user just have to turn the tip up, by pointing the stylus in the same up direction (fig. A.6(b)). To move into the bronchus and then continue the exploration, he just need to perform a forward movement as described before.



(a) *The rotation of the camera about its X-axis which allows the user to observe the top or the bottom of the scene to, in case, choose the bronchus to enter when reached a fork, as in this example is shown.*

(b) *Moving the tip of the stylus up, the user will be able to observe the top and, vice versa, moving the stylus down, he will see the bottom.*

Figure A.6: The rotation around camera Z-axis.

Bibliography

Bibliographic References

[1] EBERHARDT, R., [et al.], *Multimodality bronchoscopic diagnosis of peripheral lung lesions: a randomized controlled trial. American Journal of Respiratory and Critical Care Medicine.* Vol.176, June 2007, pp. 36−41.

[2] KIRALY, AP., [et al.], *Three-dimensional path planning for virtual bronchoscopy. IEEE Trans Med Imaging.* Vol. 23, 2004, pp. 1365−1379.

[3] SNELL, GI., [et al.] *The potential for bronchoscopic lung volume reduction using bronchial prostheses: a pilot study. Chest.* Vol. 124, 2003, pp. 1073−1080.

[4] WAHIDI, MM. , [et al.] *A Prospective Multicenter Study of Competency Metrics and Educational Interventions in the Learning of Bronchoscopy Among New Pulmonary Fellows. Chest.* Vol. 137, May 2010, pp. 137:1040−1049. Published ahead of print October 26, 2009, doi:10.1378/chest.09-1234.

[5] SENSE GRAPHICS, *HAPI Manual.* May 2009. [http://www.h3dapi.org/uploads/api/H3DAPI_2.1/docs/HAPI%20Manual.pdf, March - September 2010]

[6] BERKLEY, JJ., MIMIC TECHNOLOGIES INC. *Haptics Device. White Paper.* Seattle 2003, pp.2-4.

[7] SALISBURY, K., CONTI, F., BARBAGLI, F. *Haptic Rendering: Introductory Concepts. IEEE Computer Graphics and Applications.* Vol.24 Issue:2, March-April 2004, pp. 24-32.

[8] ZILLES, C.B., SALISBURY, J.K. *A Constraint-based God-object Method For Haptic Display. Intelligent Robots and Systems 95. 'Human Robot Interaction and Cooperative Robots', Proceedings.* Vol. 3, August 1995, pp. 146-151 vol. 3.

[9] RUSPINI, DC., KOLAROV, K., KHATIB, O. *The Haptic Display of Complex Graphical Environments. Proceedings of the 24th annual conference on Computer graphics and interactive techniques.* August 1997, pp. 245-352.

[10] CONTI, F., [et. al.] *CHAI 3D: An Open-Source Library for the Rapid Development of Haptic Scenes. IEEE World Haptics. IEEE World Haptics.* Pisa, Italy, March 2005.

[11] KITWARE *Documentation CMake 2.8.* New York, 2009. [http://www.cmake.org/cmake/help/cmake-2-8-docs.html, March - September 2010].

[12] SENSE GRAPHICS AB, *H3D Manual.* Stockholm, September 2009. [http://www.h3dapi.org/uploads/api/H3DAPI_2.1/docs/H3D%20API%20Manual.pdf, March - September 2010]

[13] NOKIA CORPORATION. *Qt Reference Documentation.* [http://doc.qt.nokia.com/4.6/, March - September 2010].

[14] BLANCHETTE, J., SUMMERFIELD, M. *C++ GUI programming with Qt 4*, Prentice Hall, 2009.

[15] SENSEGRAPHICS AB, *H3DAPI Documentation. Doxygen.* [http://www.h3dapi.org/uploads/api/H3DAPI_2.1/docs/H3DAPI/html/index.html, March - September 2010]

Complementary Bibliography

- ALBU-SHAFFER, A., HIRZINGER, G. *Cartesian Impedance Control Techniques for Torque Controlled Light-Weight Robots. Proceedings of the 2002 IEEE International Conference on Robotics and Automation.* Washington, May 2002.
- COLT, H. G., [et al.] *Simulazione di broncoscopia in realtà virtuale. CHEST italian edition.* Vol.1, 2001, pp.44-50.
- FENG-XIN, Y., [et al.] *Research of Haptic Techniques for Computer-Based Education*. Proceedings of 2009 4th International Conference on Computer Science & Education.* 2009, pp.1636-1640.
- KORY, P. D., [et al.] *Initial Airway Management skills of Senior Residents: Simulation Training Compared with Traditional Training. CHEST.* Vol.132, 2001, pp.1927-1931.
- LATHROP, R.A., [et al.] *Guidance of a Steerable Cannula Robot in Soft Tissue Using Preoperative Imaging and Conoscopic Surface Contour Sensing. 2010 IEEE International Conference on Robotics and Automation.* 2010, pp.5601-5606, The Eurographics Association and Blackwell Publishing Ltd 2003.

- LAYCOCK, S.D., DAY, A.M. *Recent Developments and Applications of Haptic Devices. Computer Graphics Forum.* Vol.22, n.2, 2003, pp.117-132, The Eurographics Association and Blackwell Publishing Ltd 2003.
- MCLENNAN, G., [et al.] *The Use of MDCT-Based Computer-Aided Pathway Finding for Mediastinal and Perihilar Lymph Node Biopsy: A Randomized Controlled Prospective Trial. Respiration.* Vol.74, 2007, pp.423-431.
- SICILIANO, B., [et al.] *Robotics. Modeling, Planning and Control.* London, Springer, 2009, cap.1 and cap.2.
- SILVESTRI, G. A. *The Evolution of Bronchoscopy Training. Respiration.* Vol.76, 2008, pp.19-20.