Development of a Software System for Virtual Fixtures Definition based on Biomedical Images, to be used in Robot-assisted Orthopedic Surgery

Francesc Trull Roca
Faculty of Informatics
Technical University of Catalonia

Final Project Report
Diploma in Computer Software

June 2012
Director: Àlicia Casals Gelpí
President: Manel Frigola Bourlon
Vocal: M. Pilar Brigida Nivela Alos
Secretary: Àlicia Casals Gelpí
Acknowledgements

My acknowledgement to Dr. Jordi Pradel Miquel, from the Department of Software Engineering and Information Systems, for welcoming me into his office and kindly answering my questions about the architecture and internal design of the developed system.

I give thanks to Dr. Vera Sacristán Adinolfi, Head of the Computational Geometry Department, for providing me with references and sources about geometric algorithms.

I also want to thank Albert Hernansanz and Joan Basomba Feliu from the Department of Automatic Control and Computer Engineering, for their guidance throughout this project.

Last and foremost, I offer my sincerest gratitude to the project director, Professor Dr. Alicia Casals in the Automatic Control and Computer Engineering Department. She has supported me throughout the project with her knowledge whilst allowing me the room to work in my own way, and has also kindly reviewed the preliminary report and the successive versions of the final report.
This project is dedicated to my mother and my sister whom have always been by my side throughout the years.
Contents:

Introduction: ........................................................................................................................................ 9

1. The context: .................................................................................................................................. 10
   1.1 Robotic surgery: ......................................................................................................................... 10
   1.2 Computer imaging ...................................................................................................................... 12
   1.3 Robots in orthopedic surgery .................................................................................................... 13
   1.4 Cranial surgery, a special case .................................................................................................. 14
   1.5 Virtual Fixtures .......................................................................................................................... 15

2. The problem: .................................................................................................................................. 16
   2.1 The complete process: ............................................................................................................... 16
   2.2 The main needs that should be addressed by virtual fixtures: .................................................. 16

3. The set-up of the project: ............................................................................................................... 18
   3.1 The segmentation tool: .............................................................................................................. 18
   3.2 The programming language: ..................................................................................................... 20
   3.3 The visualization library: ......................................................................................................... 22
   3.4 The GUI library: ....................................................................................................................... 23
   3.5 The file format: .......................................................................................................................... 24
   3.6 The XML library: ...................................................................................................................... 24
   3.7 The system architecture: ......................................................................................................... 25
   3.8 The memory management: ....................................................................................................... 26
   3.9 The build automation tool: ....................................................................................................... 27
   3.10 The Development Environment: .......................................................................................... 27

4. Requirements Analysis .................................................................................................................. 29
   4.1 Scope of the system .................................................................................................................... 29
   4.2 Definitions .................................................................................................................................. 30
      4.2.1 Case: ................................................................................................................................... 30
      4.2.2 Part: .................................................................................................................................... 30
      4.2.3 Virtual fixture: ...................................................................................................................... 31
      4.2.4 Scaled fixture: ...................................................................................................................... 31
      4.2.5 Point fixture: ........................................................................................................................ 31
      4.2.6 Line fixture: .......................................................................................................................... 31
4.2.7 Contour fixture: .................................................................................................................. 31
4.2.8 Clipping: ................................................................................................................................. 32
4.2.9 Setting: .................................................................................................................................. 32
4.2.10 Patient data: .......................................................................................................................... 32
4.3 Functional requirements ("shall lists") ...................................................................................... 32
   4.3.1 Model rendering ...................................................................................................................... 32
   4.3.2 Interaction .............................................................................................................................. 32
   4.3.3 Object visualization properties .............................................................................................. 33
   4.3.4 Scene visualization properties .............................................................................................. 33
   4.3.5 Clipping ................................................................................................................................ 33
   4.3.6 Creating cases ....................................................................................................................... 33
   4.3.7 Patient data ........................................................................................................................... 33
   4.3.8 Importing external models ..................................................................................................... 33
   4.3.9 Scaled fixtures ....................................................................................................................... 33
   4.3.10 Point fixtures ....................................................................................................................... 33
   4.3.11 Line fixtures ......................................................................................................................... 34
   4.3.12 Contour fixtures ................................................................................................................. 34
   4.3.13 Loading ............................................................................................................................... 34
   4.3.14 Saving ................................................................................................................................ 34
4.4 Nonfunctional requirements ........................................................................................................ 34
   4.4.1 Usability ................................................................................................................................ 34
   4.4.2 Reliability .............................................................................................................................. 34
   4.4.3 External Dependencies ........................................................................................................... 34
   4.4.4 Extensibility ........................................................................................................................... 34
   4.4.5 Open source .......................................................................................................................... 35
   4.4.6 Platform compatibility .......................................................................................................... 35
   4.4.7 Legal .................................................................................................................................... 35
4.5 Use case models .......................................................................................................................... 35
   4.5.1 Use Case 1 UC1: Complete process ...................................................................................... 36
   4.5.2 Use Case 2 UC2: Edit patient data ....................................................................................... 38
   4.5.3 Use Case 3 UC3: Import external part ................................................................................... 40
   4.5.4 Use Case 4 UC4: Edit scene entity’s visual properties ......................................................... 41
4.5.5 Use Case 5 UC5: Edit setting’s visual properties ........................................... 43
4.5.6 Use Case 6 UC6: Create scaled fixture .......................................................... 45
4.5.7 Use Case 7 UC7: Create point fixture ............................................................. 47
4.5.8 Use Case 8 UC8: Create line fixture ............................................................... 48
4.5.9 Use Case 9 UC9: Create contour fixture ......................................................... 50
4.5.10 Use Case 10 UC10: Edit clipper properties .................................................... 52
4.5.11 Use Case 11 UC10: Load case from disk ....................................................... 54
4.5.12 Use Case 12 UC10: Save case to disk ........................................................... 55

5. System design: ........................................................................................................ 57

5.1 System structure: ................................................................................................. 57
  5.1.1 The working case: .......................................................................................... 57
  5.1.2 Saving and loading: ....................................................................................... 58
  5.1.3 Handling user actions: .................................................................................. 60
  5.1.4 Scene entities: ............................................................................................... 61
  5.1.5 Parts: ............................................................................................................. 62
  5.1.6 Scaled fixtures: .............................................................................................. 63
  5.1.7 Point fixtures: ............................................................................................... 65
  5.1.8 Line fixtures: ................................................................................................. 67
  5.1.9 Contour fixtures: ........................................................................................... 69
  5.1.10 Clipper: ....................................................................................................... 72
  5.1.11 Setting: ....................................................................................................... 73

5.2 Class description: ................................................................................................. 75
  5.2.1 AddPartDialog: ............................................................................................ 75
  5.2.2 Case: ............................................................................................................. 75
  5.2.3 CaseCtrl: ....................................................................................................... 76
  5.2.4 CaseLoader: .................................................................................................. 76
  5.2.5 CaseSaver: .................................................................................................... 77
  5.2.6 Clipper: ........................................................................................................ 78
  5.2.7 ClipperCtrl: .................................................................................................. 78
  5.2.8 Contour fixture: ........................................................................................... 79
  5.2.9 ContourFixtureCreator: .............................................................................. 80
  5.2.10 ContourFixtureLoader: ............................................................................. 80
5.2.11 ContourFixtureProjector: ................................................................. 81
5.2.12 ContourFixtures: ........................................................................... 81
5.2.13 ContourFixtureSaver: ................................................................. 82
5.2.14 ContourFixturesCtrl: ..................................................................... 83
5.2.15 ContourFixturesEventListener: ..................................................... 84
5.2.16 ContourFixturesEventManager: ..................................................... 84
5.2.17 ContourPlaneCreator: ................................................................. 85
5.2.18 ContourWidgetCreator: ................................................................. 85
5.2.19 CreateContourFixtureDialog: ....................................................... 86
5.2.20 CreateLineFixtureDialog: ............................................................ 86
5.2.21 CreatePointFixtureDialog ........................................................... 87
5.2.22 CreateScaledFixtureDialog: ......................................................... 87
5.2.23 DefineLineFixtureCommand: ....................................................... 88
5.2.24 Line fixture: ................................................................................... 88
5.2.25 LineFixtureCreator: ....................................................................... 89
5.2.26 LineFixtureLoader: ......................................................................... 89
5.2.27 LineFixtures: ................................................................................ 90
5.2.28 LineFixtureSaver: .......................................................................... 91
5.2.29 LineFixturesCtrl: ........................................................................... 92
5.2.30 Part: ............................................................................................. 93
5.2.31 PartImporter: ................................................................................ 93
5.2.32 PartLoader: .................................................................................. 94
5.2.33 Parts: .......................................................................................... 94
5.2.34 PartSaver: ................................................................................... 95
5.2.35 PartsCtrl: ................................................................................... 96
5.2.36 PatientData: ................................................................................. 96
5.2.37 PatientDataForm: ......................................................................... 97
5.2.38 PatientDataLoader: ...................................................................... 98
5.2.39 PatientDataSaver: ....................................................................... 98
5.2.40 PlacePointFixtureCommand: ....................................................... 99
5.2.41 Point fixture: ............................................................................... 99
5.2.42 PointFixtureCreator: ................................................................... 100
5.2.43 PointFixtureLoader: ............................. 100
5.2.44 PointFixtures: .................................. 101
5.2.45 PointFixtureSaver: ................................ 102
5.2.46 PointFixturesCtrl: ............................... 103
5.2.47 PointFixturesEventListener: ....................... 104
5.2.48 PointFixturesEventManager: ...................... 104
5.2.49 ProjectorEventListener: ........................... 104
5.2.50 Scaled fixture: .................................... 105
5.2.51 ScaledFixtureCreator: ............................ 105
5.2.52 ScaledFixtureLoader: ............................. 106
5.2.53 ScaledFixtures: .................................... 106
5.2.54 ScaledFixtureSaver: ............................... 107
5.2.55 ScaledFixturesCtrl: ............................... 108
5.2.56 ScaleScaledFixtureDialog: ......................... 109
5.2.57 SceneEntitiesManager: ............................ 109
5.2.58 SceneEntitySaver: ................................ 110
5.2.59 SceneObject: ...................................... 110
5.2.60 SceneObjectLoader: .............................. 111
5.2.61 Setting: ............................................. 112
5.2.62 SettingCtrl: ......................................... 112
5.2.63 SettingLoader: .................................... 113
5.2.64 SettingSaver: ...................................... 113
5.2.65 StopDefineLineFixtureCommand: ................. 114

6. Bibliography ............................................. 115
Figures

Figure 1 The DaVinci Surgical Robot ................................................................. 10  
Figure 2 CT Scanner ....................................................................................... 12  
Figure 3 Robotic Knee Surgery ....................................................................... 13  
Figure 4 Craniotomy ......................................................................................... 14  
Figure 5 Contour to be cut on the cranium ...................................................... 17  
Figure 6 Image segmentation .......................................................................... 18  
Figure 7 Data Visualization with VTK .............................................................. 22  
Figure 8 MVC diagram ................................................................................... 25  
Figure 9 Case composition diagram ................................................................. 58  
Figure 10 Case persistence delegates .............................................................. 58  
Figure 11 CaseLoader delegates ..................................................................... 58  
Figure 12 SceneObjectLoader derived classes ................................................. 59  
Figure 13 CaseSaver delegates ....................................................................... 59  
Figure 14 SceneEntitySaver derived classes .................................................. 60  
Figure 15 CaseCtrl collaboration diagram ...................................................... 60  
Figure 16 CaseCtrl aggregation diagram ....................................................... 61  
Figure 17 SceneObject derived classes ........................................................... 61  
Figure 18 SceneEntitiesManager derived classes .......................................... 62  
Figure 19 Parts collaboration diagram ............................................................. 62  
Figure 20 PartsCtrl collaboration diagram ..................................................... 63  
Figure 21 ScaledFixtures collaboration diagram .............................................. 64  
Figure 22 ScaledFixturesCtrl collaboration diagram ...................................... 64  
Figure 23 PointFixtures collaboration diagram ............................................. 65  
Figure 24 PlacePointFixtureCommand realization diagram ............................ 65  
Figure 25 PointFixtures collaboration diagram ............................................. 66  
Figure 26 PointFixturesEventListener realizations .......................................... 66  
Figure 27 PointFixturesCtrl collaboration diagram ........................................ 67  
Figure 28 LineFixtures collaboration diagram .............................................. 67  
Figure 29 vtkCommand realizations for LineFixtures ..................................... 68  
Figure 30 LineFixturesCtrl collaboration diagram ......................................... 69  
Figure 31 ContourFixtures collaboration diagram .......................................... 69  
Figure 32 ContourFixtures collaboration diagram .......................................... 70  
Figure 33 ContourFixturesEventListener realizations ....................................... 70  
Figure 34 ProjectorEventListener realization .................................................. 71  
Figure 35 ContourFixturesCtrl collaboration diagram ..................................... 72  
Figure 36 Clipper collaboration diagram ....................................................... 72  
Figure 37 ClipperCtrl collaboration diagram .................................................. 73  
Figure 38 Setting collaboration diagram ....................................................... 73  
Figure 39 SettingCtrl collaboration diagram .................................................. 74  
Figure 40 AddPartDialog class diagram ......................................................... 75
Figure 41 Case class diagram ............................................................................................................ 75
Figure 42 CaseCtrl class diagram ........................................................................................................ 76
Figure 43 CaseLoader class diagram .................................................................................................... 77
Figure 44 CaseSaver class diagram ...................................................................................................... 78
Figure 45 Clipper class diagram ........................................................................................................... 78
Figure 46 ClipperCtrl class diagram .................................................................................................... 79
Figure 47 ContourFixture class diagram .............................................................................................. 79
Figure 48 ContourFixtureCreator class diagram ................................................................................. 80
Figure 49 ContourFixtureLoader class diagram .................................................................................... 80
Figure 50 ContourFixtureProjector class diagram ................................................................................ 81
Figure 51 ContourFixtures class diagram .............................................................................................. 82
Figure 52 ContourFixtureSaver class diagram ...................................................................................... 83
Figure 53 ContourFixturesCtrl class diagram ....................................................................................... 84
Figure 54 ContourFixturesEventListener class diagram ........................................................................ 84
Figure 55 ContourFixturesEventManager class diagram ...................................................................... 85
Figure 56 ContourPlaneCreator class diagram .................................................................................... 85
Figure 57 ContourWidgetCreator class diagram ................................................................................... 86
Figure 58 CreateContourFixtureDialog class diagram ........................................................................ 86
Figure 59 CreateLineFixtureDialog class diagram ............................................................................... 87
Figure 60 CreatePointFixtureDialog class diagram ............................................................................ 87
Figure 61 CreateScaledFixtureDialog class diagram .......................................................................... 88
Figure 62 DefineLineFixtureCommand class diagram ......................................................................... 88
Figure 63 LineFixture class diagram .................................................................................................. 89
Figure 64 LineFixtureCreator class diagram ....................................................................................... 89
Figure 65 LineFixtureLoader class diagram ......................................................................................... 90
Figure 66 LineFixtures class diagram ................................................................................................ 91
Figure 67 LineFixtureSaver class diagram .......................................................................................... 92
Figure 68 LineFixturesCtrl class diagram ........................................................................................... 92
Figure 69 Parts class diagram ............................................................................................................. 93
Figure 70 PartImporter class diagram ................................................................................................ 93
Figure 71 PartLoader class diagram .................................................................................................... 94
Figure 72 Parts class diagram .............................................................................................................. 95
Figure 73 PartSaver class diagram ....................................................................................................... 95
Figure 74 PartsCtrl class diagram ...................................................................................................... 96
Figure 75 PatientData class diagram .................................................................................................. 97
Figure 76 PatientDataForm class diagram ........................................................................................ 97
Figure 77 PatientDataLoader class diagram ...................................................................................... 98
Figure 78 PatientDataSaver class diagram ......................................................................................... 98
Figure 79 PlacePointFixtureCommand class diagram ....................................................................... 99
Figure 80 PointFixture class diagram ................................................................................................ 100
Figure 81 PointFixtureCreator class diagram ................................................................................... 100
Figure 82 PointFixtureLoader class diagram .................................................................................... 101
Figure 83 PointFixtures class diagram .............................................................................................. 102
Figure 84 PointFixtureSaver class diagram ................................................................. 103
Figure 85 PointFixturesCtrl class diagram ................................................................. 103
Figure 86 PointFixturesEventListener class diagram ...................................................... 104
Figure 87 PointFixturesEventManager class diagram ..................................................... 104
Figure 88 ProjectorEventListener class diagram ........................................................... 104
Figure 89 ScaledFixture class diagram ........................................................................... 105
Figure 90 ScaledFixtureCreator class diagram ............................................................... 105
Figure 91 ScaledFixtureLoader class diagram ............................................................... 106
Figure 92 ScaledFixtures class diagram ........................................................................... 107
Figure 93 ScaledFixtureSaver class diagram .................................................................. 108
Figure 94 ScaledFixturesCtrl class diagram .................................................................. 108
Figure 95 ScaledFixturesDialog class diagram ............................................................. 109
Figure 96 SceneEntitiesManager class diagram ............................................................. 110
Figure 97 SceneEntitySaver class diagram .................................................................... 110
Figure 98 SceneObject class diagram ............................................................................. 111
Figure 99 SceneObjectLoader class diagram ................................................................... 111
Figure 100 Setting class diagram ................................................................................. 112
Figure 101 SettingCtrl class diagram ............................................................................. 113
Figure 102 SettingLoader class diagram ........................................................................ 113
Figure 103 SettingSaver class diagram .......................................................................... 114
Figure 104 StopDefineLineFixtureCommand class diagram ......................................... 114
**Introduction:**

This report intends to reflect in a clear and understandable manner the process followed to carry out the Development of a Software System for Virtual Fixtures Definition based on Biomedical Images, to be used in Robot-assisted Orthopedic Surgery.

In the course of this report the different aspects of the project will be explained in depth to provide the reader with a complete understanding of the problem addressed, the technologies involved and the solution proposed.

The structure of this document shows how the project has been developed and implemented. The first chapter puts the project in context by introducing a series of topics to situate the reader. The second chapter justifies the motivation for the project describing the nature of the problem that will be addressed and the specific way in which it will be approached. The third chapter contains a description of all the technological tools used to carry out the present project along with the reasons why it has been decided to use them. The fourth chapter contains the requirements analysis of the system to be created. The fifth chapter contains a detailed description of the system’s internal design. The sixth chapter is the bibliography consulted throughout the project. Finally the annex contains the source code corresponding to the implementation of the system together with its documentation.
1. The context:

In the next section a series of topics are introduced to contextualize the present project. These topics are robotic surgery, computer imaging, robots in orthopedic surgery, cranial surgery and virtual fixtures.

1.1 Robotic surgery:

From their inception, surgical robots have been envisioned to extend the capabilities of human surgeons - a robot is defined as a computerized system with a motorized construction (usually an arm) capable of interacting with the environment. In its most basic form, it contains sensors, which provide feedback data on the robot’s current situation, and a system to process this information so that the next action can be determined.

Robotic systems enhance dexterity in several ways. Instruments with increased degrees of freedom greatly enhance the surgeon’s ability to manipulate instruments and thus the tissues. These systems are designed so that the surgeon’s tremor can be compensated on the end-effector motion through appropriate hardware and software filters. In addition, these systems can scale movements so that large movements of the control grips can be transformed into micro motions inside the patient.

Figure 1 The DaVinci Surgical Robot

The use of a surgical robot may or may not involve the direct role of a surgeon during the procedure; robotic surgery can thus be divided into three subcategories depending on the degree of surgeon interaction during the procedure: supervisory-controlled, telesurgical, and shared-control.
In a supervisory-controlled system the procedure is executed solely by the robot, which acts according to the computer program that the surgeon inputs into it prior to the procedure. To the moment, technology developments only allow for this procedure to be used in specific surgery activities such as making burr holes, or an incision. The surgeon is still indispensable in planning the procedure and overseeing the operation, but does not partake directly the specific activity. Because the robot performs the procedure, it must be individually programmed for it, making it extremely expensive to gather several images and data for one patient.

A telesurgical system, also known as remote surgery, requires the surgeon to manipulate the robotic arms during the procedure rather than allowing the robotic arms to work from a predetermined program. Using real-time image feedback, the surgeon is able to operate from a remote location using sensor data from the robot. Because the robot is still technically performing the procedure, it is considered a subgroup of robotic surgery. The da Vinci Surgical System, the current leading device in this field, belongs to this type of surgical robots.

The third shared-control system has the most surgeon involvement. The surgeon carries out the procedure with the use of a robot that offers steady-hand manipulations of the instrument. This enables both entities to jointly perform the tasks.
1.2 Computer imaging

Before these procedures can be carried out, robotic surgery requires the use of computer imaging to diagnose and perform the operation. These imaging modalities can generate either 3-D figures through computed tomography (CT) and magnetic resonance imaging (MRI) or 2-D ones through ultrasonography, fluoroscopy, and X-ray radiography. Out of the various methods of imaging, the main one in use is computer tomography (CT).

CT scans use back projection and detectors to obtain cross-section images that are particularly useful when diagnosing cancers and viewing the chest and the abdomen. This kind of imaging is critical to diagnosis.

However, before a surgical procedure can be carried out, there are three steps that must be overcome: planning, registration, and navigation. Planning is achieved through the careful observation of the images that are generated through these different imaging modalities. The surgeon uses this information to determine surgical pathways and methodologies. Following this step, the surgeon must coordinate the image data with the actual patient in a process known as registration. Once this is achieved, a surgeon or robot can implement the navigation step.

Using the planning and images, surgeons can manually guide instruments through the patient (computer-assisted) or robotic arms can carry out the procedure (robotic) using sensor feedback. The decision between robotic or manual navigation depends on cost, safety concerns, difficulty of execution, and other factors.
1.3 Robots in orthopedic surgery

The use of robots in surgery was at first used mainly as additional tools for enabling minimally invasive interventions. However, its use in orthopedic surgery has steadily been gaining adepts.

The main advantages of robot-assisted orthopedic surgery over conventional orthopedic techniques are improved accuracy and precision in the preparation of bone surfaces, more reliable and reproducible outcomes, and greater spatial accuracy. In fact, orthopedic surgery is ideally suited for the application of robotic systems. The ability to isolate and rigidly fix bones in known positions allows robotic devices to be securely fixed to the bone. As such, the bone is treated as a fixed object, simplifying the computer control of the robotic system. Commercially available robotic systems can be categorized as either passive or active devices, or can be categorized as positioning or milling/cutting devices.

Applications of robot-assisted orthopedic surgery currently under investigation include total hip and knee replacement, tunnel placement for reconstruction of knee ligaments, and trauma and spinal procedures. Several short-term studies demonstrate the feasibility of robotic applications in orthopedics; however, there are no published long-term data defining the efficacy of robot assisted orthopedic surgery. Issues of cost, training, and safety must be addressed before robot-assisted orthopedic surgery becomes widely available. Robot-assisted orthopedic surgery is still very much in its infancy but it has the potential to transform the way orthopedic procedures are done in the future.
1.4 Cranial surgery, a special case

Although the system developed in the present project allows the definition of virtual fixtures for any kind of orthopedic surgery it has been decided to choose the particular case of cranial surgery as a specific example. For this reason, the main cranial surgical procedures are described next.

A craniotomy is a surgical operation in which a bone flap is temporarily removed from the skull to access the brain. Related surgical procedures are craniectomy (in which the skull flap is not immediately replaced, allowing the brain to swell, thus reducing intracranial pressure) and trepanation, the creation of a burr hole through the cranium into the dura mater.

![Figure 4 Craniotomy](image)

In general, a craniotomy will be preceded by an MRI scan which provides a picture of the brain that the surgeon uses to plan the precise location for bone removal and the appropriate angle of access to the relevant brain areas. The bone flap is subsequently replaced using titanium plates and screws or another form of fixation.

This is an application in which the precise cut from surgical robots can find good use. In a shared-control system, the surgeon carries out the procedure with the use of a robot that offers steady-hand manipulations of the instrument in drilling or cutting the skull bone while assuring that underlying tissues suffer no damage.

Safety measures can be further reinforced by the implementation of virtual fixtures.
1.5 Virtual Fixtures

Traditional cooperative manipulation systems make up for the limitations of autonomous robots but the performance of such systems is still fundamentally constrained by human capabilities. Virtual fixtures, on the other hand, provide an excellent balance between autonomy and direct human control.

The term "virtual fixture" refers to a general class of guidance modes, implemented in software, that help a human-machine collaborative system (in this case, a surgical robot) perform a task by either limiting movement into restricted regions or influencing movement along desired paths. Virtual fixtures are usually haptic (force and/or tactile).

They are very beneficial for tasks that require better-than-human levels of accuracy and precision, but also require the intelligence provided by a human directly in the control loop.

Virtual fixtures attempt to capitalize on the accuracy of robotic systems while maintaining a degree of operator control, the potential benefit being safer and faster operations.

While virtual fixtures have taken a variety of forms, all can be described by two categories: guidance virtual fixtures, which assist the user in moving the robot manipulator along desired paths or surfaces, and forbidden-region virtual fixtures, which prevent the robot manipulator from entering into forbidden regions of the workspace.

The virtual fixtures find application in robot-assisted orthopedic surgery, both in directing the surgeon in his or her cut and saw paths, in drilling holes, etc., (the so-called guidance virtual fixtures), and by ensuring that the robot does not enter certain areas of the workspace, such as organ surfaces which should not be cut or delicate tissue structures (forbidden-region virtual fixtures).
2. The problem:

The objective of this project is the development of a software system to allow the definition of virtual fixtures for robot-assisted orthopedic surgery. The next section proposes a series of steps to follow to carry out the surgery and describes the main needs that should be addressed by the virtual fixtures.

2.1 The complete process:

The complete list of steps proposed in this project to carry out a robot-assisted orthopedic surgery is presented below:

1. After the proper medical examinations healthcare professionals determine that a patient needs surgery.
2. At this point it is needed to get the required medical images - using procedures such as radiology (in the wider sense), nuclear medicine, endoscopy, (medical) thermography, medical photography, etc.
3. These scanner images need then be segmented to obtain geometric models. There are many applications available for this purpose, some of them free of charge and open source. For this project, the segmentation process will be carried out using the 3dSlicer tool, a free, open source software package for visualization and image analysis.
4. The next step is importing the just-created geometric models into the virtual fixtures design system. This is the first step of the whole process that is part of the scope of the present project. The created system will have to be able to import geometric models generated with an external segmentation tool.
5. Then a qualified professional will carry out the definition of virtual fixtures on the imported models. This is the most important step concerning this project and the most part of the work done during its execution will relate to it.
6. Finally, fixtures may be imported into the robot software, getting it ready to perform surgery.

2.2 The main needs that should be addressed by virtual fixtures:

The two main actions performed by the robot during surgery are drilling holes and sawing. When it comes to drilling holes, the three variables that must be taken into account are the location of the point where the hole must be drilled, the incidence angle that the drill must adopt and the depth that the hole must have. As for sawing what must be taken into account is the outline of the path to be sawed and the depth of the cut.
Therefore, an appropriate set of virtual fixture types should be defined to assist the surgeon in the execution of the above-mentioned actions taking into account the parameters that determine them.

The developed application should provide means to create and edit all these types of virtual fixtures.
3. The set-up of the project:

The next section contains a description of all the technological tools that will be used to carry out the present project along with the reasons why it has been decided to use them. The decisions that had to be made refer to the segmentation tool used to obtain geometric models from scanner medical images, the programming language in which the software system will be implemented, the visualization library used to create, process and render geometric data, the library used to generate the Graphical User Interface, the file format used to store the system data to disk, the library used to encode and decode the system data to and from the chosen file format, the internal architecture that the developed system will present, the way in which memory management will be carried out in the system’s implementation, the tool that will be used to automate the building of the software system and the development environment that will be used to write, compile and debug the software system.

3.1 The segmentation tool:

As mentioned earlier, 3dSlicer will be the tool used to obtain geometric models from scanner medical images.

3dSlicer is a community platform created for the purpose of medical image analysis and visualization. Its features include multi-modality imaging (MRI, CT, US, nuclear medicine, and microscopy), multi organ from head to toe, bidirectional interface for devices and being expandable and interfaced to multiple toolkits.

3dSlicer provides a graphical user interface to interact with the data. In addition to manual segmentation and the creation of 3D surface models from conventional magnetic resonance images, 3dSlicer can also be used for non-rigid image registration and to incorporate models of the neurovascular bundle using image segmentation in magnetic resonance imaging-guided prostate interventions.

Figure 6 Image segmentation
3dSlicer is natively designed to be available on multiple platforms, including Windows, Linux and Mac OS X.

3dSlicer executables and source code are available under a BSD-style, free open source licensing agreement under which there are no reciprocity requirements, no restrictions on use, and no guarantees of performance. 3dSlicer leverages a variety of toolkits and software methodologies that have been labeled the NA-MIC kit.
3.2 The programming language:

For the development of any application that leverages real time three dimensional computer graphics the use of a high performance programming language is indispensable.

For this reason some very popular modern languages such as Java and C# are not good options. One characteristic of these languages is portability, which means that computer programs written in these languages will run similarly on any hardware/operating-system platform. Portability is a highly desirable feature but the particular way in which it is achieved in these languages bears a performance penalty that is unacceptable for the present purpose. To understand why this is the case let’s take a look at the way these languages function. In these languages, once the source code is ready it is compiled to an intermediate representation, instead of directly to platform-specific machine code. The intermediate representation instructions are analogous to machine code, but are intended to be interpreted by a virtual machine written specifically for the host hardware. Although this mechanism achieves the goal of high program portability the overhead of interpretation means that interpreted programs almost always run more slowly than programs compiled to native executables would.

Two languages that are widely used and don’t bear the performance penalty of the previously mentioned ones are C and C++. Therefore, these languages are found to be the most suitable for the present requirements.

C is one of the most widely used programming languages of all time. It is an imperative (procedural) systems implementation language. It is compiled using a relatively straightforward compiler and it provides low-level access to memory. It also provides language constructs that map efficiently to machine instructions and it requires minimal run-time support. Despite its low-level capabilities, the language offers cross-platform programming support. That is to say, a standards-compliant and portably written C program can be compiled for a very wide variety of computer platforms and operating systems with few changes to its source code. Some of the reasons for choosing C as the preferred programming language for the present project include its speed, stability, and near-universal availability. Other reasons why C would be a good language choice are its simplicity, code portability, and low run-time demand on system resources. Due to its thin layer of abstraction and low overhead, C allows efficient implementations of algorithms and data structures, which is very useful for programs that perform a lot of computations.

But there are also some disadvantages. Being created in the 1970’s the C programming language has a series of limitations, the most significant of them being the lack of support for object-oriented programming. C programs will often group small sections of statements into functions or subroutines each of which might perform a particular task. With this sort of designs, it is common for some of the program’s data to be global, that is to say, accessible from any part of the program. As programs grow in size, allowing any function to modify any piece of data means that bugs can have wide-reaching effects. In contrast, the object-oriented
approach encourages the programmer to place data where it is not directly accessible by the rest of the program. Instead, the data is accessed by calling specially written functions, commonly called methods. These act as the intermediaries for retrieving or modifying the data they control. The programming construct that combines data with a set of methods for accessing and managing those data is the object. It's worth mentioning that the practice of using subroutines to examine or modify certain kinds of data can also be used in non OOP languages like C but hiding the data (to prevent direct access from non-related code) is not possible.

C++ is a programming language that was devised as one approach to providing object-oriented functionality with C-like syntax. C++ adds greater typing strength, scoping, and other tools useful in object-oriented programming and permits generic programming via templates. Some of the most important characteristics of C++ include being statically typed, free-form, multi-paradigm, compiled, general-purpose and one of the most powerful languages existing. C++ is regarded as an intermediate-level language, as it comprises a combination of both high-level and low-level language features. The language began as a series of enhancements to C and progressively evolved into a full-blown independent language. The first improvement consisted in adding classes, but afterwards a large set of features where added including virtual functions, operator overloading, multiple inheritance, templates, and exception handling among others. C++ is one of the most popular programming languages and is implemented on a wide variety of hardware and operating system platforms. With an efficient compiler to native code, its application domain includes high performance software and is therefore suitable for the present project demands.

Taking into account the strengths and weaknesses of both languages it is decided to choose C++ as the preferred programming language to implement the present project. C++ will provide code portability and fast execution like C would, but also add object oriented programming support and improved memory management and type-checking capabilities.
3.3 The visualization library:

To carry out the manipulation and representation of image data the system needs to use a special-purpose library.

The libraries considered for this purpose are Open Inventor, Coin3D, Hoops 3D and VTK (the Visualization Toolkit). After reviewing comparisons of the four candidates it has been concluded that the Visualization Toolkit is the most powerful and widespread of the four and therefore it is the one chosen to be used in the development of the present project.

The Visualization Toolkit (VTK) is an open-source and freely available software system that provides functionalities for 3D computer graphics, image processing and visualization. It is written entirely in C++ and contains over six hundred different classes with 325K lines of code. VTK supports a wide variety of visualization algorithms including scalar, vector, tensor, texture, and volumetric methods. The library also supports advanced modeling techniques such as implicit modeling, polygon reduction, mesh smoothing, cutting, contouring, and Delaunay triangulation.

**Figure 7 Data Visualization with VTK**

VTK supports parallel processing, it provides an extensive information visualization framework and it integrates with various GUI toolkits such as Qt and Tk. VTK is cross-platform and runs on Linux, Windows, Mac and Unix. It provides a small suite of 3D interaction widgets and it also supports two and three-dimensional annotation. At its core VTK is implemented as a C++ toolkit and it follows the object oriented paradigm, requiring users to build applications by combining various objects into a system. VTK is used world-wide in commercial applications, research and development, and is the basis of many advanced visualization software.

The reasons for VTK being the library chosen for the present project are that, on the one hand, VTK is open-source and freely available and on the other hand it is the most complete and powerful system of its kind. It has to be mentioned though, that the VTK library has not been designed to be used for modeling purposes. Its main intent is image processing and visualization. This means that although a wide variety of algorithms are provided for the processing of image data, the tools offered to create new geometric entities are very limited. Since the present project has the necessity to generate new geometric objects from scratch.
(the virtual fixtures defined by the user) this limitation will be a significant drawback. However, because there isn’t a library available that has the characteristics of being open-source and providing advanced functionalities for both visualization and modeling, VTK remains the best suited option.

3.4 The GUI library:

The system has to provide a means for the user to interact with it, both by transmitting the commands that request the desired operations to be carried out, and receiving feedback on the effects yield by these operations. In a system of the present kind the most user-friendly way to provide this functionality is through a graphical user interface. A command line interface would be another possibility but its steep learning curve makes it less appealing. A GUI represents the information and actions available to the user through graphical icons and visual indicators such as secondary notation. The different actions in the system are performed through direct manipulation of the graphical elements in the screen.

The Visualization Toolkit provides support for two graphical user interfaces, namely Tk and Qt. Tk is open source and cross-platform. It provides a library of basic elements for building a graphical user interface and supports many different programming languages. It offers a considerable variety of widgets commonly needed to develop desktop applications. The library has been ported to run on most widely used platforms, specifically Linux, Mac OS, Unix, and Microsoft Windows. It was developed as an extension for the Tcl scripting language. Nevertheless, bindings exist for several other languages, including Ada, Perl, Python, Ruby and Common Lisp. Tk could be used in the present project through the use of an interface for the C++ programming language available under the name C++/Tk.

Qt is a cross-platform application framework that is widely used in the development of application software with a graphical user interface. Qt is developed by an open source project, the Qt Project, involving developers as individuals and from firms such as Nokia and Digia. It is written in standard C++ but makes extensive use of a special code generator called the Meta Object Compiler, or moc, together with several macros to enrich the language. It runs on a variety of desktop platforms, including Linux, Mac OS, and Windows. It has extensive internationalization support and it is free and open source software. It supports many compilers, including the GCC C++ compiler and the Visual Studio suite.

Given these two available choices, the library selected to provide the GUI related functionalities to the present project is Qt. There are a number of reasons why this library has been chosen over Tk. One of them is the fact that it is written in C++, which gives it an edge over Tk. Also, Qt is a more comprehensive library, with many more widgets available and support for greater functionality. Besides, there is a larger set of tools available for both development and debugging for the Qt GUI toolkit.
3.5 The file format:

All the data managed by the system will be stored to disk in the form of files in order to allow its recovering at a later time. The specific format used to encode the data within the file will be defined specifically for the purpose of this project. However, instead of defining a completely arbitrary set of rules, the file format used by the system will comply with the XML standard.

XML (Extensible Markup Language) is a markup language that defines a set of rules for encoding documents. The language is an open standard defined in a publicly available specification. The main characteristics that define XML are its simplicity, generality, and usability, which justifies its choice for the present purpose. It is a textual (as opposed to binary) data format and provides support for the different languages of the world via Unicode. One desirable feature of XML is the fact that its format is readable for both humans and machines. Another factor that makes it desirable is the fact that many application programming interfaces (APIs) have been developed to allow processing of XML data in different programming languages. Also, several schema systems have been created to aid in the definition of XML-based languages.

3.6 The XML library:

By using an XML-based format the application can make use of one of the many existing libraries to process the files content. The library chosen for this purpose is TinyXML.

TinyXML is a small and simple XML parser for the C++ programming language. It is free and also open source. The main strength of TinyXML is its size, as the name suggests. Even though it is limited in its capabilities it provides enough functionality for the needs of the present project while leaving a minimal footprint. The library works by parsing the XML document into a tree of node objects representing the document's contents. When storing a file to disk, the document tree is manually generated by the client code and translated by the library into XML content. Instead, when reading an XML file from disk, the library reads the file contents, loads them into memory and constructs the tree. Because it is simple and leaves a small footprint this library is a good choice for the present project. Although there are more powerful alternatives available, the extra functionalities they provide are not needed in the present case. For this reason, TinyXML is the preferred option.
3.7 The system architecture:

The architecture chosen for the software system described in the present work is the Model-View-Controller architecture. The Model-View-Controller (MVC) design assigns objects in an application one of three roles: model, view, or controller. The pattern defines not only the roles objects play in the application, but also the way objects communicate with each other. Each of the three types of objects is separated from the others by abstract boundaries and communicates with objects of the other types across those boundaries.

Model objects encapsulate the data specific to an application and define the logic and computation that manipulate and process that data. A model object can have to-one and to-many relationships with other model objects, and so sometimes the model layer of an application effectively is one or more object graphs. Much of the data that is part of the persistent state of the application (whether that persistent state is stored in files or databases) should reside in the model objects after the data is loaded into the application. Because model objects represent knowledge and expertise related to a specific problem domain, they can be reused in similar problem domains. Ideally, a model object should have no explicit connection to the view objects that present its data and allow users to edit that data—it should not be concerned with user-interface and presentation issues. User actions in the view layer that create or modify data are communicated through a controller object and result in the creation or updating of a model object. When a model object changes (for example, new data is received over a network connection), it notifies a controller object, which updates the appropriate view objects.

A view object is an object in an application that users can see. A view object knows how to draw itself and can respond to user actions. A major purpose of view objects is to display data from the application’s model objects and to enable the editing of that data. Despite this, view objects are typically decoupled from model objects in an MVC application. View objects learn about changes in model data through the application’s controller objects and communicate user-initiated changes—for example, text entered in a text field—through controller objects to an application’s model objects.

A controller object acts as an intermediary between one or more of an application’s view objects and one or more of its model objects. Controller objects are thus a conduit through which view objects learn about changes in model objects and vice versa. Controller objects can also perform setup and coordinating tasks for an application and manage the life cycles of
other objects. A controller object interprets user actions made in view objects and communicates new or changed data to the model layer. When model objects change, a controller object communicates that new model data to the view objects so that they can display it.

One of the reasons why the MVC architecture has been chosen for the present project is that it makes the domain logic independent from the user interface allowing the creation of different UIs for the same model without having to modify it. Therefore, although a Qt user interface will be created during this project, a different one (using Tk, for instance) could be easily added in the future.

### 3.8 The memory management:

In the C and C++ programming languages, allocating and freeing memory is done manually by the programmer. Memory for any data that can't be stored within a primitive data type, including objects, buffers and strings, is usually reserved on the heap. When the program no longer needs the data, the programmer frees that chunk of data with an API call. Because this process is manually controlled, human error can introduce bugs in the code. Memory leaks occur when the programmer forgets to free up memory after the program no longer needs it. Other times, a programmer may try to access a chunk of memory that has already been freed, leading to dangling pointers that can cause serious bugs or even crashes.

Programs with an automatic garbage collector (GC) try to eliminate these bugs by automatically detecting when a piece of data is no longer needed. A GC has two goals: any unused memory should be freed, and no memory should be freed unless the program will not use it anymore.

Unlike many high level languages, C++ does not impose the use of garbage collection, and mainstream C++ idioms for memory management do not assume the use of conventional automated garbage collection. The most common garbage collection method in C++ is the use of the idiom named RAII that stands for "Resource Acquisition Is Initialization". The key idea behind RAII is that a resource, whether acquired at initialization time or not, is owned by an object, and that the object's destructor will automate the release of that resource at an appropriate time. This enables C++ through RAII to support deterministic cleanup of resources, since the same approaches that work for freeing memory can also be used to release other resources (file handles, mutexes, database connections, transactions, and many more).

A particular implementation of the RAII idiom is the set of memory management smart pointers. A smart pointer is any abstract data type that simulates a pointer while providing additional features, such as automatic resource deallocation or bounds checking. These additional features are intended to reduce bugs caused by the misuse of pointers while retaining efficiency. Memory management smart pointers are those smart pointers which keep track of the objects that they point to for the purpose of memory management. As stated earlier, the misuse of pointers is a major source of bugs: the constant allocation, deallocation
and referencing that must be performed by a program written using pointers makes it very likely that some memory leaks will occur. Smart pointers try to prevent memory leaks by making the resource deallocation automatic: when the pointer to an object (or the last in a series of pointers) is destroyed, for example because it goes out of scope, the pointed object is destroyed too.

To avoid the appearance of memory leaks and dangling pointers in the software system smart pointers will be used in the development of the present project. The specific choice of smart pointers selected is the Boost Smart Pointers library. The reason for this is that many of Boost's founders are on the C++ standards committee, and the Boost Smart Pointers library has been incorporated into both Technical and the C++11 standard. Therefore, if a modern compiler is used all the necessary smart pointers will be included out of the box.

3.9 The build automation tool:

The build automation tool that will be used for the present project is CMake, the reason for this choice being that this is the tool needed to compile the VTK library.

CMake is a system for managing the build process of software using a compiler-independent method. It is cross-platform and open-source. It is designed to support directory hierarchies and applications that depend on multiple libraries. It can be used in conjunction with native build environments such as make, Apple's Xcode, and Microsoft Visual Studio. It also has minimal dependencies, requiring only a C++ compiler on its own build system. The build process with CMake takes place in two stages. First, standard build files are created from configuration files. Then the platform's native build tools are used for the actual building. Each build project contains a special file in every directory that controls the build process. While there are many built-in rules for compiling the software libraries (static and dynamic) and executables, there are also provisions for custom build rules. Some of the build dependencies can be determined automatically by CMake while the rest must be explicitly defined.

3.10 The Development Environment:

To carry out the development of the software system defined in the present work, a set of software development tools are necessary. These tools must include at least a source code editor, a compiler and a debugger. Instead of using a separate program for each necessity an integrated development environment will be used, which provides all the needed tools in a single program.

The chosen IDE for the development of the software system described in the present work is Microsoft Visual Studio.

Microsoft Visual Studio is the integrated development environment from Microsoft. It supports different programming languages by means of language services, which allow the
code editor and debugger to support nearly any programming language, provided a language-specific service exists.

Visual Studio includes a code editor that supports syntax highlighting and code completion for not only variables, functions and methods but also language constructs like loops and queries. It also includes a debugger that works both as a source-level debugger and as a machine-level debugger. It works with both managed code as well as native code and can be used for debugging applications written in any language supported by Visual Studio.

Visual C++ is Microsoft's implementation of the C++ compiler for Visual Studio. It can compile either in C mode or C++ mode. For C++, it follows the ANSI C++ spec along with a few C++0x features.

This IDE is preferred over other alternatives because it's one of the most powerful solutions available for editing, compiling and debugging C++ programs and it is available without charge for students enrolled at Technical University of Catalonia.
4. Requirements Analysis

The next section starts with a brief definition of the purpose of the required system. Then, the scope of the system is delimited specifying the functionalities that should provide. Next, a series of terms are defined. These terms conform the specific vocabulary of the application. Following is the specification of the system’s functional requirements. Then the system’s non-functional requirements are specified. Finally the set of use cases that describe the different ways in which the system may be used is presented.

Purpose of the system

The purpose of the system is to allow doctors and other professionals of the medical field to visualize parts of the body of patients and define different types of virtual fixtures on them. These virtual fixtures will be later used by a robotic system to perform surgical operations on the patient.

4.1 Scope of the system

The system will work with geometric data obtained as the result of applying a segmentation algorithm to an image volume. This volume will be obtained with one of the many medical imaging techniques available (Magnetic resonance imaging, Computed Tomography, etc.) and segmented with an external specialized application.

Then, the geometric models obtained by segmentation will be imported into the system. The system will provide the user with different tools to visualize the patient’s body parts. These tools include the possibility to define the color in which any part should be rendered and the possibility to configure the properties of the scene’s environment such as the color and intensity of the ambient lighting. Adjusting these parameters the user’s perception of the scene can be improved. The user will also be able to directly interact with the scene, translating and rotating the camera to change its position and inclination. Zooming will also be allowed, providing the user with a means to enlarge a small section of the scene and observe it in detail. The system will also offer the possibility to reduce the opacity of a selected tissue. This way the user will be able to see through it and examine its interior structures. A very powerful visualization feature that the system will provide is the possibility to clip the objects in the scene. Any object in the scene may be cut with a plane to expose its interior. The position and direction of the plane will be user-defined.

The system will provide the user with means to define two main kinds of virtual fixtures. The first kind is guidance virtual fixtures, which will assist the surgeon in moving the robot manipulator along desired paths or surfaces during surgery. The second kind is forbidden-region virtual fixtures, which will prevent the robot manipulator from entering into forbidden regions of the workspace.
The forbidden-region virtual fixtures will be implemented in the form of scaled fixtures. This type of fixture consists in a cloned part of the patient’s body that has been enlarged or shrunk.

The guidance virtual fixtures will be implemented in three different ways, namely point fixtures, line fixtures and contour fixtures. Point fixtures are useful to define the exact location of points in the patient’s body. These points may determine spots where holes should be drilled. In this scenario, line fixtures could be used to define the entry angle of the drilling device. Line fixtures define straight lines in the working space and therefore determine angles with the surfaces they intersect. Finally, contour fixtures are useful to define a path to be followed by a cutting or sawing device. These type of fixtures define arbitrary contours on the surface of the patient’s body parts.

The system will keep textual data related to the patient in order to identify the case and guide the user in the definition of the virtual fixtures.

Moreover, the system will be able to store all the data related to the case to a persistent storage device (such as a hard disk drive) to allow its retrieval later on.

4.2 Definitions

Following is the definition of a series of terms that conform the specific vocabulary of the application.

4.2.1 Case:

The case is the basic unit of work in the application. It corresponds to a medical case in the real world. It contains all the information related to a specific medical procedure programmed for a specific patient.

4.2.2 Part:

A part represents a distinguishing part of the human body. The criteria used to obtain distinct parts from the whole is not determined by the system. Parts are meant to be created outside the system and imported into it. Therefore, the software used to obtain parts (probably using segmentation algorithms) will define the used criteria. The most usual case is to distinguish different parts based on the density of the tissues that compose them. Examples of parts in the human head would be the skull bone, the gray matter, the left eyeball, etc.
4.2.3 Virtual fixture:

A virtual fixture is an entity in the working case that contains some kind of geometric information that is used to aid the surgeon during the operation.

4.2.4 Scaled fixture:

Scaled fixtures are a particular type of virtual fixtures which are created by cloning the geometry of an existing part of the patient’s body and then modifying its size.

These fixtures are very useful to protect a part in the working case by enwrapping it with an enlarged version of itself. In this way, the scaled fixture defines a forbidden zone for the surgeon to enter and prevents the wrapped part of being damaged.

Another use for scaled fixtures is to limit the allowed depth of a cut or a hole in a specific part by creating a clone of the part and shrinking it appropriately so that its surface determines the maximum depth the surgeon is allowed to reach.

4.2.5 Point fixture:

Point fixtures are a particular type of virtual fixtures which define points on the surface of specific parts of the patient’s body.

These fixtures can be used to determine with precision the location where holes should be drilled in the patient’s body.

4.2.6 Line fixture:

Line fixtures are a particular type of virtual fixtures which define straight lines ending in point fixtures of the working case.

These fixtures are very useful when it comes to specify the entry angle of a drill boring a hole into a body part.

In this scenario, the location of the hole could be determined by a point fixture while the entry angle of the drill could be defined with a line fixture.

4.2.7 Contour fixture:

Contour fixtures are a particular type of virtual fixtures which define arbitrary contours drawn on the surface of body parts.
These fixtures are very useful to define a path that will guide a robotic device when sawing or cutting a body part. For instance, it’s usual in brain surgery to cut a window in the patient’s scull and access the brain through it. The outline of this window, whichever its shape might be, can be precisely defined with a contour fixture.

4.2.8 Clipping:

Clipping is a visualization technique consisting in cutting an object (or a group of objects) with an oriented plane so that its interior is exposed and can be inspected.

This is a very powerful tool used in the system to give the user an improved understanding of the morphology of the elements present in the scene.

4.2.9 Setting:

The setting is the environment where the objects in the scene are placed. It involves concepts such as the color and intensity of the light illuminating them and the color of the scene’s background.

4.2.10 Patient data:

Patient data is the data related to the patient in the working case. This data includes the patient id, his first name, middle name and last name, the gender and age, the diagnosis of his condition, any relevant medical history and the required treatment. A code to identify the case is also included.

4.3 Functional requirements ("shall lists")

4.3.1 Model rendering

The system shall be able to render 3-dimensional models in the screen

4.3.2 Interaction

The system shall enable interaction with the scene (translate the camera, rotate the camera, zoom in/zoom out ...)
4.3.3 Object visualization properties

The system shall allow the user to change the visualization properties of any object in the scene (its color, transparency ...)

4.3.4 Scene visualization properties

The system shall allow the user to change the visualization properties affecting the scene as a whole (the background color, light intensity ...)

4.3.5 Clipping

The system shall allow the user to clip the objects in the scene and inspect its interior.

4.3.6 Creating cases

The system shall allow the user to create new cases.

4.3.7 Patient data

The system shall allow the user to introduce and edit patient data in the working case.

4.3.8 Importing external models

The system shall allow the user to import external models of body parts to the working case.

4.3.9 Scaled fixtures

The system shall allow the user to generate scaled fixtures. This type of fixture is shaped like one of the parts in the case but have a different size.

4.3.10 Point fixtures

The system shall allow the user to generate point fixtures. This type of fixture consists in a point placed on the surface of one of the parts in the case.
4.3.11 Line fixtures

The system shall allow the user to generate line fixtures. This type of fixture consists in a straight line that goes from a user defined point in the scene to one of the point fixtures in the case.

4.3.12 Contour fixtures

The system shall allow the user to generate contour fixtures. This type of fixture consists in a user defined contour placed on the surface of one of the parts in the case.

4.3.13 Loading

The system shall allow the user to load all data associated with a case from disk.

4.3.14 Saving

The system shall allow the user to save all data associated with a case to disk.

4.4 Nonfunctional requirements

4.4.1 Usability

The system shall be easy to use and fast to learn by a non-technical user.

4.4.2 Reliability

The system shall not crash when errors occur in the interaction with the hardware. Instead, it shall show a informing error message to the user and continue normal operation.

4.4.3 External Dependencies

The system shall only depend on the following three external libraries: Qt, VTK and TinyXML.

4.4.4 Extensibility

The system shall be highly extensible and allow addition of new functionality in the future.
4.4.5 Open source

The system shall follow the open source philosophy. All the generated code shall be freely available.

4.4.6 Platform compatibility

The system shall work in Windows, Mac OSX and Linux Operating Systems.

4.4.7 Legal

The system shall be developed as part of a technical engineering final project in Universitat Politècnica de Catalunya. The legal rights relating to the system shall be the ones described in the regulations of the university.

4.5 Use case models

Following are presented the set of use cases that describe the different ways in which the application may be used.
4.5.1 Use Case 1 UC1: Complete process

Scope:
VFDesigner application

Level:
User goal

Primary actor:
User

Stakeholders and interests:
- Surgeon: wants the robot to aid him during the surgical operation.
- User: wants to define the appropriate fixtures to assist the surgeon.
- Patient: wants the surgical operation to be safe and successful.
- Hospital: wants its processes to be performed with the maximum effectiveness and the minimum cost.
- Government: wants to assure safety proceedings are followed.

Preconditions:
User has launched VFDesigner.

Success guarantee:
The case contains all the desired information and is saved successfully to disk.

Main success scenario or basic flow:
1. The user starts a new case.
2. The user introduces all the information related to the patient (Edit patient data).
3. The user imports an external part (Import external part).
4. The system displays the visual representation of the part in the screen.
5. The user sets the part’s visualization properties to improve its perception and ease its manipulation (Edit scene entity’s visual properties)
6. The user repeats steps 3-5 until all parts are added to the scene.
7. The user sets the scene’s visualization properties (Set setting’s visual properties).
8. The user creates a virtual fixture (Create scaled fixture, Create point fixture, Create line fixture, Create contour fixture).
9. The system displays the visual representation of the virtual fixture in the main window and its properties in the proper tab.
10. The user edits the virtual fixture’s visual properties (Edit scene entity’s visual properties)
11. The user repeats steps 8-10 until all desired fixtures are created.
12. The user saves the case to disk (Save case to disk).

Alternative flows:

*a. At any time the user wants to make sure the work done is not lost.
   1. The user saves the working case to disk (Save case to disk).

*b. At any time the user decides to work in another case.
   1. The user closes the current working case.
   2. The user creates a new case or loads an existing one (Load case from disk).

1a. The user had previously created the case and saved it to disk.
   1. The user loads the previously saved case (Load case from disk).
   2. The system displays all the parts and virtual fixtures in the scene.
   3. The user performs the pending operations (if any) and saves the changes.

2a. The user doesn’t have all the needed information at the moment.
   1. The user skips the introduction of the data associated with the case.
   2. The user performs the desired operations.
   3. At a later moment, the missing information is introduced (Edit patient data).
4.5.2 Use Case 2 UC2: Edit patient data

Scope:
VFDesigner application

Level:
User goal

Primary actor:
User

Stakeholders and interests:
- User: wants the working case to contain all the needed patient data.

Preconditions:
The user has created or opened a working case.

Success guarantee:
The proper patient data is associated with the case.

Main success scenario or basic flow:
1. The user asks the system to edit the patient data.
2. The system presents the proper screen for the purpose.
3. The user introduces the case identifier.
4. The user introduces the patient’s identifier.
5. The user introduces the patient’s first name.
6. The user introduces the patient’s middle name.
7. The user introduces the patient’s last name.
8. The user introduces the patient’s gender.
9. The user introduces the patient’s age.
10. The user introduces the patient’s diagnosis.
11. The user introduces the patient’s relevant medical history.
12. The user introduces the patient’s required treatment.
13. The user tells the system that the data input is completed.

Alternative flows:
3-12a. The user wants to introduce the data in a specific order.

1. The user introduces the data in whichever order prefers.
3-12b. The user wants to introduce only part of the data.

1. The user introduces only part of the data.
2. The user switches to another task.
3. Possibly at a future moment, the user introduces the rest of the data.
4.5.3 Use Case 3 UC3: Import external part

**Scope:**
VFD designer application

**Level:**
User goal

**Primary actor:**
User

**Stakeholders and interests:**
- User: wants to import an external part to the working case.

**Preconditions:**
The user has created or opened a working case.
Parts have been created with an external application.

**Success guarantee:**
An external part stored in a file is imported into the case.

**Main success scenario or basic flow:**
1. The user asks the system to import an external part.
2. The system asks the user to select the file containing the external part.
3. The user selects the desired file.
4. The system imports the external part into the case.

**Alternative flows:**
*a. The user changes his opinion about importing an external part
   1. The user asks the system to cancel the current operation.
4.5.4 Use Case 4 UC4: Edit scene entity’s visual properties

**Scope:**
VFDesigner application

**Level:**
User goal

**Primary actor:**
User

**Stakeholders and interests:**
- User: wants to edit the visual properties of a scene entity in the working case.

**Preconditions:**
The working case contains at least one scene entity.

**Success guarantee:**
The scene entity’s visual properties are modified to satisfy the intent of the user.

**Main success scenario or basic flow:**
1. The user selects the scene entity whose visual properties wants to change.
2. The user asks the system to modify the selected scene entity’s color.
3. The system asks the user to specify the new color.
4. The user picks his preferred color for the selected scene entity.
5. The system changes the selected scene entity’s color to the picked one.
6. The user asks the system to modify the selected scene entity’s opacity.
7. The system asks the user to specify the new level of opacity.
8. The user chooses the level of opacity he prefers for the selected scene entity.
9. The system modifies the selected scene entity’s opacity level to the one picked by the user.
10. The user defines whether the selected scene entity should be visible in the scene or not.
11. The system modifies the appearance of the scene to include or not the scene object accordingly.

12. The user defines whether the selected scene entity should be clipped in the scene or not.

13. The system modifies the appearance of the scene to clip or not the scene object accordingly.

14. The user defines whether all scene entities of the current type should be visible in the scene or not.

15. The system modifies the appearance of the scene to include or not all the scene objects of the current type.

16. The user defines whether all scene entities of the current type should be clipped in the scene or not.

17. The system modifies the appearance of the scene to clip or not all the scene objects of the current type.

Alternative flows:

*a. The user wishes to edit the scene entity’s properties in a different order.

1. The user edits the scene entity’s properties in his preferred order.

*b. The user doesn’t want to edit some of the scene entity’s properties.

1. The user edits only the scene entity’s properties that want to modify.
4.5.5 Use Case 5 UC5: Edit setting’s visual properties

**Scope:**
VFDesigner application

**Level:**
User goal

**Primary actor:**
User

**Stakeholders and interests:**
- User: wants to edit the visual properties of the setting in the working case.

**Preconditions:**
A working case has been created or loaded from disk.

**Success guarantee:**
The setting’s visual properties are modified to the taste of the user.

**Main success scenario or basic flow:**
1. The user tells the system that it wants to edit the setting’s properties.
2. The system presents the user the appropriate screen for the task.
3. The user picks his preferred color for the main background color.
4. The system changes the main background color to the selected one.
5. The user picks his preferred color for the secondary background color.
6. The system changes the secondary background color to the selected one.
7. The user chooses whether he wants a gradient in the background or not.
8. The system modifies the appearance of the scene to reflect the user’s choice.
9. The user picks his preferred color for the light illuminating the scene.
10. The system modifies the appearance of the objects in the scene according to the color of the light illuminating them.
11. The user selects the intensity of the light illuminating the scene.

12. The system modifies the appearance of the objects in the scene according to the intensity of the light illuminating them.

*Alternative flows:

*a. The user wants to edit the setting’s properties in a different order.

1. The user edits the setting’s properties in whichever order prefers.

*b. The user doesn’t want to edit some of the setting’s properties.

1. The user edits only the setting’s properties that he wants to modify.
4.5.6 Use Case 6 UC6: Create scaled fixture

**Scope:**
VFDesigner application

**Level:**
User goal

**Primary actor:**
User

**Stakeholders and interests:**
- User: wants to create a new scaled fixture and add it to the working case.

**Preconditions:**
There is at least one part in the working case.

**Success guarantee:**
A scaled fixture with the desired properties has been created and added to the working case.

**Main success scenario or basic flow:**
1. The user tells the system that it wants to create a new scaled fixture.
2. The system presents the user the appropriate screen for the task.
3. The user introduces a name for the new fixture.
4. The user selects a part from the working case to be used as a source for the new fixture.
5. The user tells the system to create a new scaled fixture with the introduced parameters.
6. The system creates a new fixture with the name introduced by the user and the same shape as the part selected by the user.
7. The user tells the system to scale the created fixture.
8. The system asks the user to specify the scaling ratio.
9. The user experiments with different ratios.
10. The system enlarges or shrinks the visual representation of the selected fixture accordingly.
11. The user tells the system to use the current ratio as definitive

12. The system scales the geometry of the fixture with the chosen ratio.

**Alternative flows:**

3a. There is a scaled fixture in the working case with the name introduced by the user.

   1. The system informs the user about it and asks for a different name.
   2. The user introduces a different name for the new fixture.

7-12a. The user doesn’t want to scale the fixture.

   1. The user skips steps 7-12 and the fixture maintains its original size.

12b. The user is not content with the size of the fixture.

   1. The user repeats steps 7-12 until the desired size for the fixture is obtained.
4.5.7 Use Case 7 UC7: Create point fixture

**Scope:**
VFDesigner application

**Level:**
User goal

**Primary actor:**
User

**Stakeholders and interests:**
- User: wants to create a new point fixture and add it to the working case.

**Preconditions:**
There is at least one part in the working case.

**Success guarantee:**
A point fixture with the desired properties has been created and added to the working case.

**Main success scenario or basic flow:**
1. The user tells the system that it wants to create a new point fixture.
2. The system presents the user the appropriate screen for the task.
3. The user introduces a name for the new fixture.
4. The user selects the part from the working case where the point will be placed onto.
5. The user picks the exact location for the point fixture.
6. The system creates a new point fixture with the name introduced by the user in the specified location.

**Alternative flows:**
3a. There is a point fixture in the working case with the name introduced by the user.
   1. The system informs the user about it and asks for a different name.
   2. The user introduces a different name for the new fixture.
4.5.8 Use Case 8 UC8: Create line fixture

Scope:
VFDesigner application

Level:
User goal

Primary actor:
User

Stakeholders and interests:
- User: wants to create a new line fixture and add it to the working case.

Preconditions:
There is at least one point fixture in the working case.

Success guarantee:
A line fixture with the desired properties has been created and added to the working case.

Main success scenario or basic flow:
1. The user tells the system that it wants to create a new line fixture.
2. The system presents the user the appropriate screen for the task.
3. The user introduces a name for the new fixture.
4. The user selects the point fixture from the working case that will define the ending point of the line fixture.
5. The user defines the location of the starting point of the line fixture.
6. The system creates a new line fixture with the name introduced by the user and the proper ending and starting points.

Alternative flows:
3a. There is a line fixture in the working case with the name introduced by the user.
   1. The system informs the user about it and asks for a different name.
   2. The user introduces a different name for the new fixture.
6a. The user is not content with the starting point of the line fixture.

1. The user asks the system to edit the starting point of the selected line fixture.
2. The user defines the new location for the starting point of the line fixture.
4.5.9 Use Case 9 UC9: Create contour fixture

Scope:
VFDesigner application

Level:
User goal

Primary actor:
User

Stakeholders and interests:
- User: wants to create a new contour fixture and add it to the working case.

Preconditions:
There is at least one part in the working case.

Success guarantee:
A contour fixture with the desired properties has been created and added to the working case.

Main success scenario or basic flow:
1. The user tells the system that it wants to create a new contour fixture.
2. The system presents the user the appropriate screen for the task.
3. The user introduces a name for the new fixture.
4. The user selects the part from the working case where the contour fixture will be placed onto.
5. The system displays a plane upon the chosen part.
6. The user creates a new node to define the shape of the contour.
7. The user moves the node to the desired location on the plane.
8. The user deletes the nodes that does not need.
9. The user repeats steps 9-11 until he obtains the desired shape.
10. The user tells the system that the current shape is the definitive.
11. The system creates a new contour fixture with the specified name and a shape consisting of the projection of the contour in the plane on the surface of the selected part.

*Alternative flows:*

3a. There is a contour fixture in the working case with the name introduced by the user.

1. The system informs the user about it and asks for a different name.
2. The user introduces a different name for the new fixture.
4.5.10 Use Case 10 UC10: Edit clipper properties

Scope:
VFDesigner application

Level:
User goal

Primary actor:
User

Stakeholders and interests:
- User: wants to edit the clipper’s properties to visualize the interior of the objects in the scene.

Preconditions:
There is an open working case in the system.

Success guarantee or post-conditions:
The objects in the scene have been clipped in the defined manner and their interior can be inspected.

Main success scenario or basic flow:
1. The user defines as clippable the objects in the scene that wishes to inspect (Edit scene objects visual properties).
2. The user tells the system that it wants to edit the clipper’s properties.
3. The system presents the user the appropriate screen for the task.
4. The user selects the desired plane orientation (the available options are axial, sagittal and coronal).
5. The system changes the orientation of the plane to the one selected by the user.
6. The user selects which of the two sides of the plane is the clipping one.
7. The system deletes from the scene the parts of the objects located at the selected side of the clipping plane.
8. The user selects the position of the plane along the axis orthogonal to it.
9. The system moves the plane to the defined location.

*Alternative flows:

*a. The user wishes to edit the clipper’s properties in a different order.

2. The user edits the clipper’s properties in his preferred order.

*b. The user doesn’t want to edit some of the clipper’s properties.

2. The user edits only the clipper’s properties that wants to modify.
4.5.11 Use Case 11 UC10: Load case from disk

Scope:
VFDesigner application

Level:
User goal

Primary actor:
User

Stakeholders and interests:
- User: wants to load into the system a working case previously saved to disk.

Preconditions:
There is at least one working case stored to disk.

Success guarantee:
The saved working case is successfully loaded into the system.

Main success scenario or basic flow:
1. The user tells the system to load a working case from disk.
2. The system asks the user to select the desired saved case in the file system.
3. The user selects the case he want to load into the system from the file system.
4. The system loads into the system all the information related to the selected case.

Alternative flows:
4a. There is a hardware or operating system error during the loading operation.
   1. The system informs the user about the circumstance and cancels the ongoing operation.
   2. The system remains functional and doesn’t crash.
4.5.12 Use Case 12 UC10: Save case to disk

**Scope:**
VFDesigner application

**Level:**
User goal

**Primary actor:**
User

**Stakeholders and interests:**
- User: wants to save all the information related to the current working case to disk.

**Preconditions:**
There is one working case in the system.

**Success guarantee:**
All the information related to the current working case has been saved to disk.

**Main success scenario or basic flow:**

1. The user tells the system to save the current working case to disk.

2. The system asks the user to introduce a name for the working case and a location in the file system to place it.

3. The user introduces the name for the working case and selects his preferred file system location for it.

4. The system stores all the information related to the current working case to disk in the selected location and under the specified name.

**Alternative flows:**

3a. There is already a working case stored in the file system in the same location and with the same name.

    1. The system informs the user about the circumstance and asks whether it should overwrite the existing case or cancel the operation.
    2. The user chooses one of the two proposed options.
    3. The system overwrites the existing case or cancels the operation according to the option specified by the user.
4a. There is a hardware or operating system error during the saving operation.

3. The system informs the user about the circumstance and cancels the ongoing operation.
4. The system remains functional and doesn’t crash.
5. System design:

Due to the complexity of a software system such as the present it is not advisable to describe its design in a single narration. Instead, it is more useful to present a series of descriptions, each corresponding to a different level of abstraction.

In the present case three different levels of abstraction can be clearly distinguished.

The first level sees the classes that constitute the system as basic entities whose particular interfaces are unknown. At his level of abstraction the focus is on the relations among the constituent modules without concern for the properties of each module. The first part of the next section describes the system at this level of abstraction and provides the reader with an overall understanding of its internal organization.

The second level increases the detail and looks at each class individually while describing its internal structure. The second part of the next section presents a description of the system at this level of abstraction. Each class is briefly described and the reader learns about the properties it holds and the services it provides.

The third level corresponds to the implementation of the different classes and their operations in the programming language. This implementation can be consulted in the annex, which contains all the source code of the system with its documentation.

5.1 System structure:

5.1.1 The working case:

All the domain concepts in the application are included inside the general concept of a case. A case represents a real world medical case and is represented in the system by the same named class. A case is composed by parts, scaled fixtures, point fixtures, line fixtures, contour fixtures, patient data, clipper and setting. Each one of these represents a specific concept in the system and will be explained in detail later on.

The next illustration shows all the sub-modules that compose the case:
5.1.2 Saving and loading:

The case can be loaded from and saved to persistent storage. To achieve this goal the class uses two specialized helper classes, as can be seen in the following diagram.

Since the case contains a considerable number of different entities, each of them with its own particular properties, the case loader class delegates the task of loading each kind of object to a specialized helper. Therefore the case loader’s duty is to coordinate the activities of these helpers to achieve the complete loading of a case. The next figure shows the dependency diagram of the case loader.
Since they all have the duty of loading a particular type of scene object to the working case, all specific loaders share some common functionality. For this reason they are implemented with the inheritance hierarchy depicted in the following figure.

![SceneObjectLoader derived classes](image)

**Figure 12** SceneObjectLoader derived classes

Like the case loader, the case saver delegates the task of saving each specific type of entity to a specialized helper class. The next diagram shows the dependency relationships between the case saver and its delegates.

![CaseSaver delegates](image)

**Figure 13** CaseSaver delegates

All the specialized savers have the duty of saving a particular type of scene object to disk. Therefore, there is some common functionality among them too.

They are also implemented with an inheritance hierarchy but in this case the base class is a template class since each derived class needs to deal with a different type of entity.

The following figure presents the aforementioned situation.
5.1.3 Handling user actions:

The case controller is the main controller in the system. Its responsibility is to handle all the events triggered by the view as a consequence of actions performed by the user. These user actions are translated into the appropriate system actions and the proper domain service requests are carried out.

The case controller class directly handles the actions create a new case, open an existing case, close the current case and save the current case to disk. It’s also responsible for the actions edit patient data and show VFDesigner information. The following diagram illustrates the relations between the case controller and the other modules involved in these system actions.

For all the other actions that can be performed in the system, case controller counts with the help of various delegates. Each delegate is responsible for handling a group of related events.
All the events within a particular group are triggered by a singular tab in the view. To facilitate the use of the application the view is designed in such a way that all the interactions with the user relating to a specific concept in the domain are performed through the same tab. The next diagram shows the aggregation of specialized controllers contained in the case controller.

![CaseCtrl aggregation diagram](image)

**Figure 16 CaseCtrl aggregation diagram**

### 5.1.4 Scene entities:

Scene entities are those objects that have a representation in the scene. These are parts, scaled fixtures, point fixtures, line fixtures and contour fixtures. Even though each particular type of object has its unique characteristics they all share a common set of attributes and operations. For this reason they are implemented in the system with an inheritance hierarchy as can be seen in the following figure:

![SceneObject derived classes](image)

**Figure 17 SceneObject derived classes**

The management of all the objects of the same type is carried out by a specialized manager. Since all the distinct types of objects share some common functionality its managers will do likewise. Each manager maintains a collection containing the entities it owns and these entities are of a different type in each case. For this reason, the base manager class is a template class. The class has a type parameter named entity which will be used with the proper particular class for each instantiation.
The next diagram shows the inheritance relationship of the different entity managers in the system.

**Figure 18 SceneEntitiesManager derived classes**

### 5.1.5 Parts:

The parts class contains an aggregation of all the part objects in the system. A part object represents a distinguishing part of the human body. What exactly constitutes a distinguishing part is defined by the user but possible examples would be the skull, the gray matter or the optic tract. Unlike the rest of elements in the scene parts are not defined in the application but imported from external sources. To import an existent part into the system the parts class depends on part importer. As scene elements parts must be added to the setting in order to be rendered in the scene. For this purpose the parts instance maintains a reference to the setting. Other than that all the elements in the scene can be clipped to allow the user to visualize its interior. Therefore the parts class also contains a reference to the clipper. The aforementioned relations between parts and its collaborators are represented in the following figure:

**Figure 19 Parts collaboration diagram**
The parts controller is responsible for mediating between the view and the parts. The information related to the parts is presented to the user in a particular section of the view named partsTab. Also, the geometric representation of the parts is displayed with the rest of the scene in the render window. The user interacts with the proper part of the view and his actions trigger a series of events which are handled by the pertinent controller. In this case, the user can select an item from the list and the tab will be updated to display its attributes. Through a series of widgets these attributes can be edited to one’s liking. To import an external part to the working case, a suited dialog is presented to the user to allow the introduction of a name for the new part and the selection of the file containing the desired geometric data. This dialog is managed by a specialized class named “add part dialog”.

The following diagram shows the relations between the parts controller and its collaborating modules.

![PartsCtrl collaboration diagram](image)

5.1.6 Scaled fixtures:

The scaled fixtures class contains an aggregation of all the scaled fixture objects in the scene. A scaled fixture is a particular kind of fixture consisting of a part shaped entity whose size has been modified. This type of fixture allows the definition of a wrapper around a protected part by enlarging a clone of it. It can also define limits at how deep a hole can be drilled or a cut can be performed by shrinking the clone.

This class delegates the creation of a new scaled fixture to a specialized helper. It contains a reference to the parts instance because it needs access to it so that the user can specify which part to use as a source for a new fixture. The references to the clipper and the setting objects are used for the same reasons mentioned in the parts class. In the next figure the relations between the scaled fixtures class and its immediate dependencies are represented.
The scaled fixtures controller acts as a mediator between the view and the scaled fixtures. Each scaled fixture has a geometric representation displayed in the render window with the rest of the scene. The scaledFixturesTab is the particular section of the view where the information related to the scaled fixtures is presented to the user. The user interacts with the tab and as a result a series of events are triggered which are handled by the scaled fixtures controller. Every time the user selects an item from the list the tab is updated to display its attributes. Interacting with the appropriate widgets these attributes can be consulted and modified. To create a scaled fixture the user must select a part from the working case and introduce a name for the new fixture. This is done through a specialized dialog managed by the class “create scaled fixture dialog”. Once the fixture is added to the working case the user can change its size through a different dialog controlled by the class “scale scaled fixture dialog”.

Figure 21 ScaledFixtures collaboration diagram

Figure 22 ScaledFixturesCtrl collaboration diagram
5.1.7 Point fixtures:

The point fixtures class contains an aggregation of all the point fixture objects in the scene. A point fixture is a particular type of fixture that allows the user to specify points upon the surface of parts in the scene. These points can be used, for example, to determine the exact location where a drilling should be performed. The creation of a new point fixture is delegated to a specialized class. The point fixtures class holds a reference to the parts instance to allow the user to specify which part to place the point onto. It also holds references to setting and clipper for the same reason as scaled fixtures and parts.

![PointFixtures collaboration diagram](image1)

The actual location of the point fixture is determined by the user through interaction with the rendering window. Once the user has initiated the point fixture creation process by specifying a name for the new fixture and selecting a part from the case to place the point onto, it determines the exact location of the point with a mouse click inside the rendering window. This action triggers the proper VTK event which is handled by place point fixture command. This class implements the library defined interface vtkCommand and it carries out the task of translating the coordinates of the mouse cursor in the screen to the world coordinates in the scene and requesting the creation of the new fixture with the calculated values.

![PlacePointFixtureCommand realization diagram](image2)
When the user defines the location of the point fixture by clicking the mouse with the cursor situated over the desired position a new point fixture object is created and added to the case (and to the scene). At this point the controller should update its appearance to include the new item. To inform the controller about this fact so that it can perform the proper actions in response, the observer pattern is used. The class point fixtures event manager is responsible for this duty. It holds the collections of registered listeners (and provides the service to add and remove them during execution) and it announces the occurrence of the events by sending the proper message to each listener by means of the specified interface.

![Figure 25 PointFixtures collaboration diagram](image)

Point fixtures controller subscribes itself to the events of interest using two particular implementations of the point fixtures event listener interface. These implementation classes are named “On point fixture added” and “on point fixture removed”.

![Figure 26 PointFixturesEventListener realizations](image)

The point fixtures controller is the intermediary between the view and the point fixtures. Each point fixture is represented in the render window as a tiny sphere placed on the surface of a part. The attributes of the point fixtures are available to the user through the pointFixturesTab. As a result of the user interacting with the tab a series of events are triggered which are handled by the point fixtures controller. When the user selects an item from the list the tab is updated to display its attributes. Interacting with the appropriate widgets these attributes can be consulted and modified. To create a point fixture the user must select a part from the working case and introduce a name for the new fixture. This is done through a specialized dialog managed by the class “create point fixture dialog”.

![Diagram](image)
5.1.8 Line fixtures:

The line fixtures class contains the set of all the line fixture objects in the working case. A line fixture object is, as its name implies, a particular kind of fixture that is shaped as a straight line. These fixtures can be used, for instance, to specify the entry angle of a drill boring a hole into a body part. The line is defined by a starting point and an ending point. The latter is chosen as one of the point fixtures in the scene while the former is specified by the user with the mouse cursor.
The position of the starting point is defined with the mouse cursor and therefore must change constantly during mouse motion. The class responsible for achieving this function is “define line fixture command”. Every time the mouse cursor position changes this class calculates the coordinates of the new starting point and requests line fixtures to update the fixture’s state and representation accordingly. Once the desired position is achieved the user orders the system to freeze the current shape of the fixture with a mouse click. The class responsible for this action is “stop define line fixture command”. These two classes are specific implementations of the vtkCommand interface offered by the VTK library as the means to define event listeners.

The line fixtures controller is the intermediary between the view and the line fixtures. Each line fixture is represented in the render window as a straight line with the defined color and width. The attributes of the line fixtures can be consulted and edited in the lineFixturesTab. When the user interacts with the tab the appropriate events are triggered and the point fixtures controller is responsible for handling them. When the user selects an item from the list the tab is updated to display its attributes. Interacting with the appropriate widgets these attributes can be consulted and modified. To create a line fixture the user must select a point fixture from the working case and introduce a name for the new fixture. This is done through a specialized dialog managed by the class “create line fixture dialog”.

![Class Diagram]

*Figure 29 vtkCommand realizations for LineFixtures*
5.1.9 Contour fixtures:

The contour fixtures class contains an aggregate with all the contour fixtures in the working case. A contour fixture is a particular kind of fixture characterized by its contour shape. The contour can be open or closed and can have any desired shape. This type of fixture can be used to define an outline to be cut or sawed. The fixture is first defined on the surface of a virtual plane and latter projected onto a chosen part.

The contour fixtures class delegates the creation of the virtual plane on the surface of which the contour will be defined to the specialized class contour plane creator. The actual definition of the contour is performed using a specially designed widget which is managed by another helper module named contour widget creator. Once the desired shape is achieved each point
in the contour is projected to the surface of the chosen part in order to obtain the desired fixture. The projection process is managed by another delegate called contour fixture projector.

When the definition process is finished and the new fixture is created and added to the case the contour fixtures controller needs to update its appearance to include the new item. Of course the contour fixtures class is a domain entity and should not be coupled to the controllers in any way. For this reason the announcement of the addition (and removal) of a new fixture in the case is performed by means of the observer pattern. The implementation of this functionality is conducted by a specialized module named contour fixtures event manager. This class manages the collection of listeners to be notified when the proper event occurs. It provides operations to register an unregister listeners to a specific event they might be interested in, and it communicates the occurrence of the event to all interested parties when it takes place. “On contour fixture added” and “on contour fixture removed” are the two particular implementations of the contour fixtures event listener interface used by the contour fixtures controller to register its interest in the so called events.
In order to be able to give feedback to the user through the display of a progress bar, the contour fixtures controller needs to be constantly informed about the state of the projection process. For this reason the contour fixture projector provides an event publishing functionality. Once the projection begins the module announces the fact to all registered listeners. Another event is published every time a point is projected and a final event is announced when the projection process has concluded. Since the three events are intimately related the three correspondent listeners are bundled together in a single interface named projector event listener. The class projector bindings is the implementation of this interface used to inform the contour fixtures controller about the occurrence of the events.

The contour fixtures controller is the intermediary between the view and the contour fixtures. Each contour fixture is represented in the render window as a contour drawn on the surface of a part. The attributes of the contour fixtures can be consulted and edited in the contourFixturesTab. When the user interacts with the tab the appropriate events are triggered and the contour fixtures controller is responsible for handling them. When the user selects an item from the list the tab is updated to display its attributes. Interacting with the appropriate widgets these attributes can be consulted and modified. To create a contour fixture the user must select a part from the working case and introduce a name for the new fixture. This is done through a specialized dialog managed by the class “create contour fixture dialog”.

![Class Diagram](image-url)
5.1.10 Clipper:

The clipper class is responsible for managing the scene’s clipping plane. All the objects represented in the scene can be clipped by this plane in order to allow the user to visualize their interior. The plane’s position and orientation are defined by the user to obtain the desired effect. The plane can adopt any of the three canonical orientations used in the medical field: axial, sagittal and coronal. The plane’s location can be moved throughout the scene along the main axes. When the plane intersects an object, the part of the object situated on one side of the plane stays intact while the part of the object placed in the other side of the plane is erased from the scene. The user can define which of the two sides of the plane will clip the scene and which one won’t. The following figure shows the need for all scene entity managers to hold a reference to the clipper to allow its constituent elements to be clipped.
the user interacts with the tab the appropriate events are triggered and the clipper controller is responsible for handling them and propagating the proper service requests to the clipper.

![ClipperCtrl collaboration diagram](image)

**Figure 37 ClipperCtrl collaboration diagram**

### 5.1.11 Setting:

The setting class is responsible for managing the visual properties of the scene. Every object must be added to the scene through the setting class in order to be rendered.

The class also manages the resulting appearance of the setting by controlling the background colors, the existence of a gradient and the color and intensity of the light illuminating the scene. The following diagram shows the need for every scene entity manager to hold a reference to the setting in order to add and remove object to the scene.

![Setting collaboration diagram](image)

**Figure 38 Setting collaboration diagram**

The setting controller is the intermediary between the view and the setting. The attributes of the setting can be consulted and modified through a series of widgets in the settingTab. When
the user interacts with the tab the appropriate events are triggered and the setting controller is responsible for handling them and propagating the proper service requests to the setting.

Figure 39 SettingCtrl collaboration diagram
5.2 Class description:

5.2.1 AddPartDialog:

AddPartDialog manages the dialog to add an imported part to the case. The user must introduce the name for the new part and pick the path to the file containing the geometric data in vtk format.

![AddPartDialog class diagram](image)

Figure 40 AddPartDialog class diagram

5.2.2 Case:

Case represents the entire working case. It contains references to the different types of information relating to the case (Parts, ScaledFixtures, PointFixtures, LineFixtures, ContourFixtures, PatientData, Setting and Clipper). It is also able to save and load itself from a set of files using CaseSaver and CaseLoader.

![Case class diagram](image)

Figure 41 Case class diagram
5.2.3 CaseCtrl:

CaseCtrl is the main controller in VFDesigner. It manages the application’s main window and it’s directly responsible for handling the user actions to create a new case, open an existing case from a file, close the working case and save the working case to disk. It also handles the actions edit patient data and get VFDesigner information. The rest of the user actions are handled by other controllers (PartsCtrl, ScaledFixturesCtrl, PointFixturesCtrl, LineFixturesCtrl, ContourFixturesCtrl, ClipperCtrl and SettingCtrl) which in turn are managed by this class.

![CaseCtrl class diagram](image)

**Figure 42 CaseCtrl class diagram**

5.2.4 CaseLoader:

CaseLoader is a class responsible for loading an entire case from disk. The actual loading of each specific kind of object in the case is performed by a suited class (PartLoader, ScaledFixturesLoader, PointFixturesLoader, LineFixturesLoader, ContourFixturesLoader, ClipperCtrl and SettingCtrl) and all of them are managed by CaseLoader. Each geometric entity is stored in a different vtp file and the rest of the information related to the case is stored in the vfd (an XML-based ad hoc format) file.
5.2.5 CaseSaver:

CaseSaver is a class responsible for saving an entire case to disk. The actual saving of each specific kind of object in the case is performed by a suited class (PartSaver, ScaledFixtureSaver, PointFixtureSaver, LineFixtureSaver, ContourFixtureSaver, PatientDataSaver and SettingSaver) and all of them are managed by CaseSaver. Each geometric entity is stored in a different vtp file and the rest of the information related to the case is stored in the vfd (an XML-based ad hoc format) file.
5.2.6 Clipper:

Clipper is the class that manages the clipping plane. All the objects represented in the scene can be clipped by the plane. The plane’s orientation can be changed to be axial, sagittal or coronal. The plane’s position can be changed along the scene’s bounding box. The plane can also be inverted so that the clipped part of the scene changes accordingly.

5.2.7 ClipperCtrl:

ClipperCtrl is the controller responsible for handling the user actions related to the clipper. The class is responsible for capturing the events triggered as a result of actions performed by the
user interacting with the clipperTab and translating them into the appropriate requests to be send to the clipper object. It is also responsible for updating the representation of the clipperTab to reflect the changes in the state of the clipper.

![Figure 46 ClipperCtrl class diagram](image_url)

### 5.2.8 Contour fixture:

ContourFixture is the class that represents one contour fixture in the scene. The fixture has a unique identifier and an actor that is its visual representation. The visual properties of the fixture can be read and written through appropriate methods. These properties include its color, width, visibility and whether the fixture is clipped or not. This class inherits part of its functionality from SceneObject.

![Figure 47 ContourFixture class diagram](image_url)
5.2.9 ContourFixtureCreator:

ContourFixtureCreator is the class responsible for creating a new contour fixture. Its interface consists in a single method which receives the fixture id, the contour's geometric data and a reference to the clipping plane and creates a new ContourFixture object with the appropriate values. This class is used by ContourFixtures every time it needs to create a new ContourFixture object.

![Figure 48 ContourFixtureCreator class diagram](image)

5.2.10 ContourFixtureLoader:

ContourFixtureLoader is a CaseLoader helper class that provides the functionality to create a new ContourFixture and set its state with the values read from files. The geometry of the contour is read from a vtp file and the rest of its properties are read from the vfd file. Part of the functionality for this class is inherited from SceneObjectLoader.

![Figure 49 ContourFixtureLoader class diagram](image)
5.2.11 ContourFixtureProjector:

ContourFixtureProjector is the class responsible for projecting the plane's defined contour on the surface of the specified scene part. The contour is defined by the user on the surface of a virtual plane to allow fast editing. Once the desired shape is obtained the contour is projected point by point onto the three dimensional surface of the chosen part. This process is computationally costly and takes some time. For this reason, the class publishes events at the start of the projection, each time a point is projected and at the end of the projection to give the user feedback about the progress being made.

![ContourFixtureProjector class diagram](image)

5.2.12 ContourFixtures:

ContourFixtures is the class responsible for managing all the contour fixtures in the working case. This class receives service requests from ContourFixturesCtrl. These requests include the definition and creation of new fixtures and requests to obtain or modify the value of a property in a specific fixture. This class inherits part of its functionality from SceneEntitiesManager.
5.2.13 ContourFixtureSaver:

ContourFixtureSaver is a CaseSaver helper class that provides the functionality to save a ContourFixture to persistent storage. The geometry of the contour is stored in a vtp file while the rest of the data related to the fixture is stored in the vfd file. This class inherits part of its functionality from SceneEntitySaver.
5.2.14 ContourFixturesCtrl:

ContourFixturesCtrl is the controller responsible for handling the user actions related to contour fixtures. The class is responsible for capturing the events triggered as a result of actions performed by the user interacting with the contourFixturesTab and translating them into the appropriate requests to be send to the ContourFixtures object. It is also responsible for updating the appearance of the contourFixturesTab to display the attributes of the selected fixture.
Development of a Software System for Virtual Fixtures Definition based on Biomedical Images, to be used in Robot-assisted Orthopedic Surgery

5.2.15 ContourFixturesEventListener:

ContourFixturesEventListener is an interface that allows the creation of specific listeners that can be registered for ContourFixtures events and notified whenever these events occur.

5.2.16 ContourFixturesEventManager:

ContourFixturesEventManager is a ContourFixtures helper class responsible for managing the collection of listeners to be notified when the proper event occurs. It provides operations to register an unregister listeners to a specific event they might be interested in, and it communicates the occurrence of the event to all interested parties when it takes place.
5.2.17 ContourPlaneCreator:

ContourPlaneCreator is the class responsible for creating the virtual plane on the surface of which the user will define the shape of the contour fixture that will later be projected on the three-dimensional surface of the chosen part. The plane is orthogonal to the orientation of the camera and it is positioned a certain distance in front of the camera in the direction of the focal point.

5.2.18 ContourWidgetCreator:

ContourWidgetCreator is the class responsible for creating the ContourWidget that will allow the user to define the shape of a contour on the surface of the contour plane. The contour is defined by means of a variable number of nodes united by curve lines.
5.2.19 CreateContourFixtureDialog:

CreateContourFixtureDialog is the class responsible for managing the dialog to create a new contour fixture. The user must introduce a name for the new fixture and choose which part should the defined contour be projected onto. The class validates that the input is correct and displays an appropriate error message otherwise.

5.2.20 CreateLineFixtureDialog:

CreateLineFixtureDialog is the class responsible for managing the dialog to create a new line fixture. The user must introduce a name for the new fixture and choose which point should be used as the line's end point. The line's start point will be defined with the mouse cursor. The class validates that the input is correct and displays an appropriate error message otherwise.
5.2.21 CreatePointFixtureDialog

CreatePointFixtureDialog is the class responsible for managing the dialog to create a new point fixture. The user must introduce a name for the new fixture and choose which part should the new point fixture be placed upon. The user will define the exact location of the point by performing a mouse click action over the desired location in the chosen part. The class validates that the input is correct and displays an appropriate error message otherwise.

5.2.22 CreateScaledFixtureDialog:

CreateScaledFixtureDialog is the class responsible for managing the dialog to create a new scaled fixture. The user must introduce a name for the new fixture and choose which part should be cloned to become the new scaled fixture. Afterwards the user will be able to scale (make either bigger or smaller) the fixture to achieve the desired size. The class validates that the input is correct and displays an appropriate error message otherwise.
5.2.23 DefineLineFixtureCommand:

DefineLineFixtureCommand is the class that represents the command that will be executed every time the mouse cursor changes its position during the definition of a line fixture. The command translates the display coordinates of the cursor to the world coordinates of the scene and requests LineFixtures to update the fixture being defined appropriately.

5.2.24 Line fixture:

LineFixture is the class that represents one line fixture in the working case. One end of the line (the end point) corresponds to one point fixture in the scene. The other end (the start point) is defined by the user with the mouse cursor. Like the other scene objects the line fixture’s visualization properties can be both consulted and modified.
5.2.25 LineFixtureCreator:

LineFixtureCreator is the class responsible for creating a new line fixture. Its interface consists of one method that receives as parameters all the necessary information (the fixture’s id, its start and end positions and the scene's clipping plane) and returns the newly created fixture with the appropriate attributes.

5.2.26 LineFixtureLoader:

LineFixtureLoader is a CaseLoader helper class that provides the functionality to create a new LineFixture and set its state with the values read from a file. The geometry of the LineFixture only consists of two points and therefore it’s stored in the vfd file with the other properties of the fixture. Part of the functionality for this class is inherited from SceneObjectLoader.
5.2.27 LineFixtures:

LineFixtures is the class responsible for managing all the line fixture objects in the working case. The class receives requests from the LineFixturesCtrl which in turn receives them from the user actions. This class manages the creation of new line fixture objects and the edition of the existing ones. It also propagates requests to consult or modify one property of a specific line fixture to the appropriate object. This class inherits part of its functionality from SceneEntitiesManager.
5.2.28 LineFixtureSaver:

LineFixtureSaver is a CaseSaver helper class that provides the functionality to save a LineFixture to persistent storage. The geometry of the fixture only consists of two points and therefore it’s stored in the vfd file with the other properties of the fixture. This class inherits part of its functionality from SceneEntitySaver.
5.2.29 LineFixturesCtrl:

LineFixturesCtrl is the controller responsible for handling the user actions related to line fixtures. The class is responsible for capturing the events triggered as a result of actions performed by the user interacting with the lineFixturesTab and translating them into the appropriate requests to be send to the LineFixtures object. It is also responsible for updating the appearance of the lineFixturesTab to display the attributes of the selected fixture.
5.2.30 Part:

Part is the class that represents one part in the working case. It is structurally and functionally identical to a SceneObject but it has its own definition because it represents a different concept.

![Part class diagram](image)

Figure 69 Parts class diagram

5.2.31 PartImporter:

PartImporter is the class responsible for importing an external part into the working case. This part might be created with some segmentation software and stored as a vtk polyData file. The interface of the class consists of one method that receives as parameters the id of the new part, the path of the file containing the polyData and the scene’s clipping plane and returns a newly created part with the appropriate attributes.

![PartImporter class diagram](image)

Figure 70 PartImporter class diagram
5.2.32 PartLoader:

PartLoader is a CaseLoader helper class that provides the functionality to create a new Part and set its state with the values read from files. The geometry of the Part is read from a vtp file and the rest of its properties are read from the vfd file. Part of the functionality for this class is inherited from SceneObjectLoader.

![PartLoader class diagram](image)

Figure 71 PartLoader class diagram

5.2.33 Parts:

Parts is the class responsible for managing all the part objects in the working case. The class receives requests from the PartsCtrl which in turn receives them from the user actions. This class manages the creation of new part objects and the edition of the existing ones. It also propagates requests to consult or modify one property of a specific part to the appropriate object. This class inherits part of its functionality from SceneEntitiesManager.
5.2.34 PartSaver:

PartSaver is a CaseSaver helper class that provides the functionality to save a Part to persistent storage. The geometry of the part is stored in a vtp file while the rest of the data is stored in the vfd file. This class inherits part of its functionality from SceneEntitySaver.
5.2.35 PartsCtrl:

PartsCtrl is the controller responsible for handling the user actions related to parts. The class is responsible for capturing the events triggered as a result of actions performed by the user interacting with the partsTab and translating them into the appropriate requests to be send to the parts object. It is also responsible for updating the appearance of the partsTab to display the attributes of the selected part.

![PartsCtrl class diagram](image)

5.2.36 PatientData:

PatientData is the class responsible for managing all the patient data relating to the working case. This data include the case id, the patient id, the first name, middle name and last name of the patient, the patient's gender and age, the diagnosis for his condition, any relevant information in the patient's medical history and the proposed treatment.
5.2.37 PatientDataForm:

PatientDataForm is the class responsible for managing the form containing all the information related to the patient in the working case. This form allows both consulting and modifying of the different fields displaying the pertinent data. The class retrieves the patient information from the PatientData object and updates the PatientData object when the user requests it.
5.2.38 **PatientDataLoader:**

PatientDataLoader is a CaseLoader helper class that provides the functionality to create a new PatientData object containing the data read from the case file. All the information is read from the vfd file containing the case data.

![PatientDataLoader class diagram](image)

5.2.39 **PatientDataSaver:**

PatientDataSaver is a CaseSaver helper class that provides the functionality to save a PatientData object to persistent storage. All the relevant data in the object is stored in the vtp file that represents the working case.

![PatientDataSaver class diagram](image)
5.2.40 PlacePointFixtureCommand:

PlacePointFixtureCommand is the class that represents the command that will be executed when the user performs a mouse click action during the definition of a point fixture. The command obtains the coordinates of the point where the ray cast orthogonal to the view plane intersects the surface of the chosen part. Then it requests PointFixtures to create a new point fixture positioned at that location.

![PlacePointFixtureCommand class diagram](image)

**Figure 79 PlacePointFixtureCommand class diagram**

5.2.41 Point fixture:

PointFixture is the class that represents one point fixture in the working case. The position of the fixture is defined by the user by performing a mouse click action at the desired location upon the chosen part. Like the other scene objects the line fixture's visualization properties can be both consulted and modified.
5.2.42 PointFixtureCreator:

PointFixtureCreator is the class responsible for creating a new point fixture. Its interface consists in a single method which receives the fixture id, the point geometric location and a reference to the clipping plane and creates a new PointFixture object with the appropriate values. This class is used by PointFixtures every time it needs to create a new PointFixture object.

5.2.43 PointFixtureLoader:

PointFixtureLoader is a CaseLoader helper class that provides the functionality to create a new PointFixture and set its state with the values read from files. The position of the point and all
its representational properties are read from the vfd file. Part of the functionality for this class is inherited from SceneObjectLoader.

5.2.44 PointFixtures:

PointFixtures is the class responsible for managing all the point fixture objects in the working case. The class receives requests from the PointFixturesCtrl which in turn receives them from the user actions. This class is responsible for the creation of new point fixture objects and their addition to the working case. It also propagates requests to consult or modify one property of a specific point fixture to the appropriate object. This class inherits part of its functionality from SceneEntitiesManager.
5.2.45 PointFixtureSaver:

PointFixtureSaver is a CaseSaver helper class that provides the functionality to save a PointFixture to persistent storage. The geometry of the fixture only consists of one point and therefore it's stored in the vfd file with the other properties of the fixture. This class inherits part of its functionality from SceneEntitySaver.
5.2.46 PointFixturesCtrl:

PointFixturesCtrl is the controller responsible for handling the user actions related to point fixtures. The class is responsible for capturing the events triggered as a result of actions performed by the user interacting with the pointFixturesTab and translating them into the appropriate requests to be send to the PointFixtures object. It is also responsible for updating the appearance of the pointFixturesTab to display the attributes of the selected fixture.
5.2.47 PointFixturesEventListener:

PointFixturesEventListener is an interface that allows the creation of specific listeners that can be registered for PointFixtures events and notified whenever these events occur.

![PointFixturesEventListener class diagram](image)

5.2.48 PointFixturesEventManager:

PointFixturesEventManager is a PointFixtures helper class responsible for managing the collection of listeners to be notified when the proper event occurs. It provides operations to register an unregister listeners to a specific event they might be interested in, and it communicates the occurrence of the event to all interested parties when it takes place.

![PointFixturesEventManager class diagram](image)

5.2.49 ProjectorEventListener:

ProjectorEventListener is an interface that allows the creation of specific listeners that can be registered for ContourFixtureProjector events and notified whenever these events occur.

![ProjectorEventListener class diagram](image)
5.2.50 **Scaled fixture:**

ScaledFixture is the class that represents one scaled fixture in the working case. The shape of the fixture is defined by cloning an existing part. Its size, however, can be modified by the user to fit its intended purpose. Like the other scene objects the scaled fixture's visualization properties can be both consulted and modified. This class inherits part of its functionality from SceneObject.

![Figure 89 ScaledFixture class diagram](image)

5.2.51 **ScaledFixtureCreator:**

ScaledFixtureCreator is the class responsible for creating a new scaled fixture. Its interface consists of one method that receives as parameters all the necessary information (the fixture's id, the part whose geometry will be cloned and the scene's clipping plane) and returns the newly created fixture with the appropriate attributes.

![Figure 90 ScaledFixtureCreator class diagram](image)
5.2.52 ScaledFixtureLoader:

ScaledFixtureLoader is a CaseLoader helper class that provides the functionality to create a new ScaledFixture and set its state with the values read from files. The geometry of the ScaledFixture is read from a vtp file and the rest of its properties are read from the vfd file. Part of the functionality for this class is inherited from SceneObjectLoader.

![ScaledFixtureLoader class diagram](image)

5.2.53 ScaledFixtures:

ScaledFixtures is the class responsible for managing all the scaled fixture objects in the working case. The class receives requests from the ScaledFixturesCtrl which in turn receives them from the user actions. This class manages the creation of new scaled fixture objects and the edition of the existing ones. It also propagates requests to consult or modify one property of a specific scaled fixture to the appropriate object. This class inherits part of its functionality from SceneEntitiesManager.
5.2.54 ScaledFixtureSaver:

ScaledFixtureSaver is a CaseSaver helper class that provides the functionality to save a ScaledFixture to persistent storage. The geometry of the ScaledFixture is stored in a vtp file while the rest of the data is stored in the vfd file. This class inherits part of its functionality from SceneEntitySaver.
ScaledFixturesCtrl is the controller responsible for handling the user actions related to scaled fixtures. The class is responsible for capturing the events triggered as a result of actions performed by the user interacting with the scaledFixturesTab and translating them into the appropriate requests to be send to the ScaledFixtures object. It is also responsible for updating the appearance of the scaledFixturesTab to display the attributes of the selected fixture.
5.2.56 ScaleScaledFixtureDialog:

ScaleScaledFixtureDialog is the class responsible for managing the dialog to scale an existing scaled fixture. The user can move a slider to adjust the scaling factor from half the current size to twice the current size. The appearance of the fixture in the scene is scaled accordingly allowing the user to preview the resulting effect. Once the user achieves the intended size and clicks the accept button, the actual geometry of the fixture is modified to reflect the requested scaling transformation.

![Figure 95 ScaledFixturesDialog class diagram](image)

5.2.57 SceneEntitiesManager:

SceneEntitiesManager is the template class that serves as a base class for all the classes that manage some type of scene entities (Parts, ScaledFixtures, PointFixtures, LineFixtures and ContourFixtures). It contains the implementation of the operations common throughout all the specific entity managers. These include adding and removing an entity to the collection, checking if there is an entity with a given id, retrieving a entity with a given id, obtain a list of all the entity ids, consult and modify the color, the opacity, the visibility and the condition of being clipped. The visibility and the condition of being clipped can also be set for all the entities at once.
5.2.58 SceneEntitySaver:

SceneEntitySaver is a template class that serves as the base class for all the classes that save a scene entity (PartSaver, ScaledFixtureSaver, PointFixtureSaver, LineFixtureSaver and ContourFixtureSaver). It provides the common structure and behaviour among the these classes.

5.2.59 SceneObject:

SceneObject is the class that serves as a base class for all the classes representing one type of object in the scene (Part, ScaledFixture, PointFixture, LineFixture and ContourFixture). It
contains the implementation of the operations common throughout all the specific scene objects. These include consulting and modifying the id, the actor, the color, the opacity, the visibility and the condition of being clipped.

![SceneObject class diagram](image)

**Figure 98 SceneObject class diagram**

5.2.60 SceneObjectLoader:

SceneObjectLoader is a class that serves as the base class for all the classes that load a scene object (PartLoader, ScaledFixtureLoader, PointFixtureLoader, LineFixtureLoader and ContourFixtureLoader). It provides the common structure and behaviour among the these classes.

![SceneObjectLoader class diagram](image)

**Figure 99 SceneObjectLoader class diagram**
5.2.61 Setting:

Setting is the class responsible for managing the properties of the scene as a whole. It offers operations to consult and modify the main background color, the secondary background color, whether the background of the scene is plane or is a gradient, the color quality of the light illuminating the scene and its intensity. It also offers operations to add and remove actors to the scene and to obtain its geometric boundaries.

5.2.62 SettingCtrl:

SettingCtrl is the controller responsible for handling the user actions related to the setting. The class is responsible for capturing the events triggered as a result of actions performed by the user interacting with the settingTab and translating them into the appropriate requests to be send to the setting object. It is also responsible for updating the representation of the settingTab to reflect the changes in the state of the setting.
5.2.63 SettingLoader:

SettingLoader is a CaseLoader helper class that provides the functionality to create a new Setting object containing the data read from the case file. All the information is read from the vfd file containing the case data.

5.2.64 SettingSaver:

SettingSaver is a CaseSaver helper class that provides the functionality to save a Setting object to persistent storage. All the relevant data in the object is stored in the vtp file that represents the working case.
5.2.65 StopDefineLineFixtureCommand:

StopDefineLineFixtureCommand is the class that represents the command that will be executed when the user has finished defining the shape of the line fixture. At this point, the user has placed the mouse cursor in the desired position for the line fixture starting point and has performed a mouse click action. The command requests LineFixtures to stop defining the shape of the fixture and to set the current one as definitive.
6. Bibliography


Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John Design patterns: Elements of Reusable Object-Oriented Software Addison-Wesley Professional, 1995.


Alexandrescu, Andrei Modern C++ Design: Generic Programming and Design Patterns Applied Addison-Wesley Professional, 2001

Schroeder, Will; Martin, Ken; Lorensen, Bill The Visualization Toolkit, An Object-Oriented Approach to 3D Graphics Third Edition Kitware, Inc. 2002

Schroeder, Will The VTK User’s Guide Kitware, Inc. 1998