Datateknik C, Examensarbete, 15 högskolepoäng

A VISUALIZATION TOOL FOR DRILL RIG SIMULATORS USED IN SOFTWARE DEVELOPMENT

Mikael Larsson

Dataingenjörsprogrammet 180 hp

Örebro vårterminen 2010

Examinator: Mathias Broxvall

ETT VISUALISERINGSVERKTYG FÖR BORRIGSSIMULATORER ATT ANVÄNDA I MJUKVARUUTVECKLING

Örebro universitet
Institutionen för teknik
701 82 Örebro

Örebro University
Department of Technology
SE-701 82 Örebro, Sweden

Abstract

Boomer is a machine that is developed and produced by Atlas Copco Rock Drills AB, which is used for underground mining and tunneling. It is a blast-hole drilling rig equipped with drills that are attached to the arms, called booms, which the rig holds. The machine is controlled and monitored by Atlas Copco's Rig Control System (RCS), which consists of a number of intelligent units connected in a CAN-net. When developing software for the RCS, a simulator that makes it possible to run the software on an ordinary desktop PC is used. The problem is that there is no intuitive way to see how the booms are oriented, while positioning. Therefore it is desirable to have a 3D visualization of the rig, with focus on the booms, which can be used alongside the simulator to get immediate feedback about the movements of the booms. This report describes the process of developing an application that handles communication with the simulator and the 3D visualization.

Sammanfattning

Boomer är en maskin som utvecklas och produceras av Atlas Copco Rock Drills AB. Maskinen används vid gruvbrytning och tunnelkonstruktion. Boomer är en spränghålsborrigg som är utrustad med borrar vilka är monterade på riggens armar, kallade bommar. En Boomer övervakas och kontrolleras av Atlas Copcos kontrollsystem, RCS, som är ett system bestående av intelligenta enheter sammankopplade i ett CAN-nät. Vid utveckling av mjukvara till RCS används en simulator som gör det möjligt att köra mjukvaran på en vanlig PC. Problemet är att det inte finns något intuitivt sätt att se hur bommarna är riktade medans de blir positionerade. Därför är det önskvärt med en 3D visualisering av borriggen, med fokus på dess boomar, som kan användas tillsammans med simulatorn för att ge en direkt återkoppling av boomarnas förflyttning. Denna rapport beskriver utvecklingsprocessen för en applikation som hanterar kommunikationen med simulatorn och 3D visualiseringen.

Acknowledgements

I would like to thank everyone at the Automation Department at Atlas Copco Rock Drills AB, especially Thorsten Michelfelder and Kreso Milic for their supervision during this work. I also like to thank my supervisor Jack Pencz at Örebro University.

Mikael Larsson

Table of contents

1 Introduction

Atlas Copco Rock Drills AB develops and produces equipment that is used in mining (both over- and underground) and tunneling industries.

Boomer is a machine type that is developed and produced by Atlas Copco and used for underground mining and tunneling. The Boomer is a blast-hole drilling rig, equipped with drills that are attached onto each of the one or more arms, called booms, which the machine holds. Figure 1.1 shows a Boomer with three booms and one service platform. The machine drills between 20 and 150 holes, depending on size, with a length of 4 - 5 meters. These holes are then filled with explosives. After a detonation, other machines are used to transport the rocks, remove loose rocks and secure the mountain. The booms can be controlled manually, in which an operator steers the booms and drills using an operator panel, or automatically, in which a drilling sequence is loaded.

Figure 1.1: A Boomer XE3C

The Boomer is controlled and monitored by Atlas Copco's Rig Control System (RCS). This system consists of a variable number of units depending on the size and complexity of the machine that has to be controlled. Each unit is a computer that is connected to the other units via a CAN-net.

1.1 Problem description

The Automation Department at Atlas Copco develops the software called RCS, used on the drill rigs. To avoid only being able to test software on actual hardware simulation is used. A simulator that runs on a desktop PC simulates the CAN-net and its connected units. The problem with the simulation is that there is no intuitive way to see how the booms move. There is a 2D image that shows the position of the drill bit, but to see how the rest of the boom moves you have to look at and interpret several sensor values. A real-

time 3D visualization of the booms, which can be used alongside the simulator, would give the developers immediate feedback on how the software affects movements of the booms. An application that does this visualization is desirable and creating it is the objective of this ex-job.

The following requirements are defined to be fulfilled by the application:

- Be able to communicate with the simulator in order to retrieve sensor information,
- correctly visualize the booms with 3D graphics,
- show boom motion using the retrieved sensor information,
- be both scalable and configurable to support multiple booms and different boom setups (including boom type, feeder type and length).

1.2 Results

The results of this work were an application that communicates with the simulator to receive signal data, which is used together with 3D graphics to visualize the movements of a boom. It supports multiple booms and different boom setups. A screenshot of the final application is shown in figure 1.2.

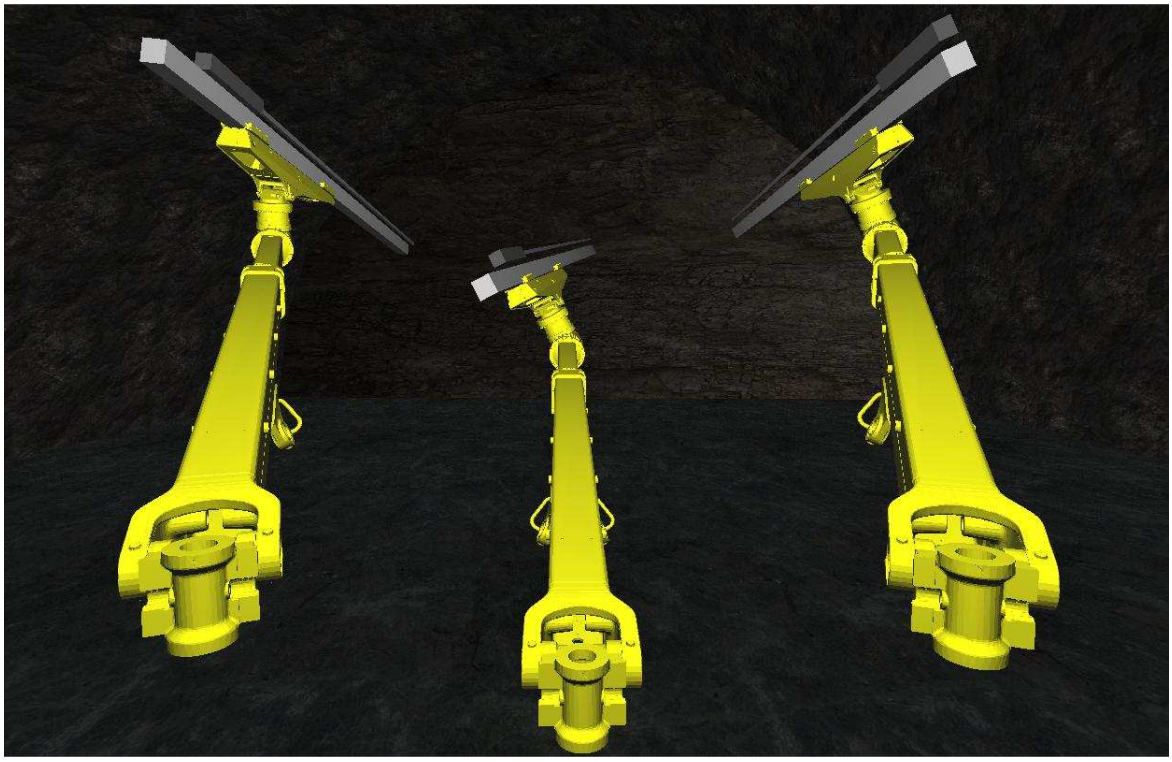
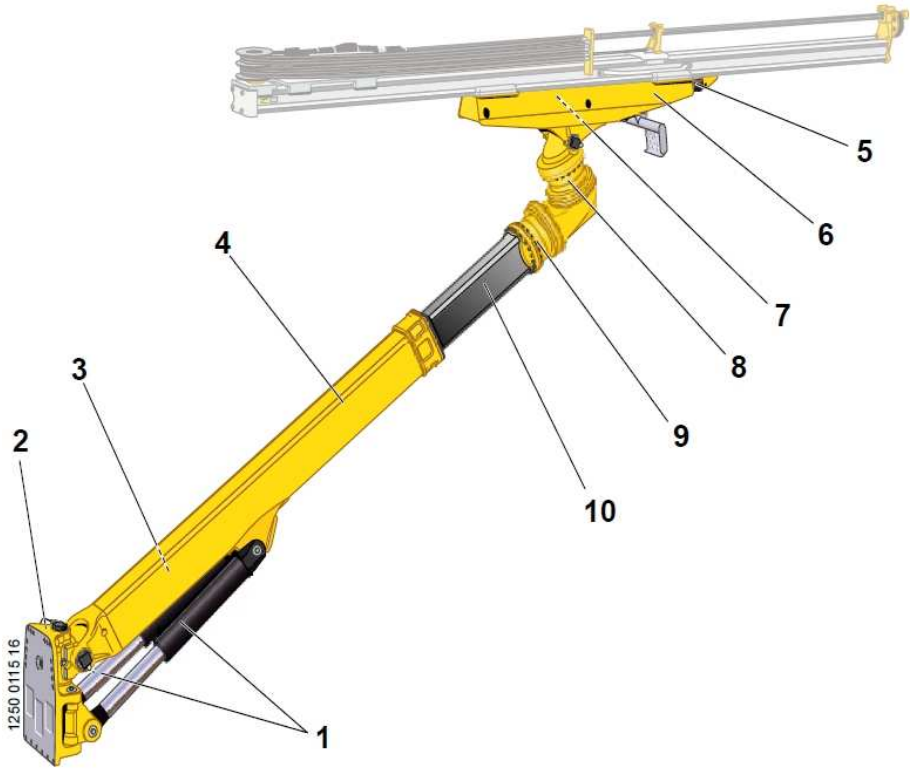


Figure 1.2: Screenshot of the 3D visualization application

2 Background

2.1 The booms

A boom consists of several parts that are connected to each other. Each part has an axis which it rotates around or translates along. The movements are actuated by hydraulic cylinders that are mounted on the boom. Figure 2.1 shows an illustration of a boom, pointing out the main parts.



1. Boom cylinders.
2. Boom console.
3. Extension cylinder.
4. Boom body.
5. Feed extension cylinder.
6. Feed holder.
7. Stick cylinder (used to tilt the feeder)
8. Feed swing.
9. Feed rotation.
10. Boom extension.

Figure 2.1: Illustration of a BUT 45

The geometry of a boom is described by a link model. Using the nomenclature from the field of robotics, a boom is a set of rigid links connected together at various joints. A joint is either revolute, which mean that it allows a relative rotation between two links, or prismatic, which mean that it allows a linear relative motion between two links. A link model defines the length and direction of each link and around, or along, which axis each joint rotates, or moves. Figure 2.2 shows a link model for a common boom type.

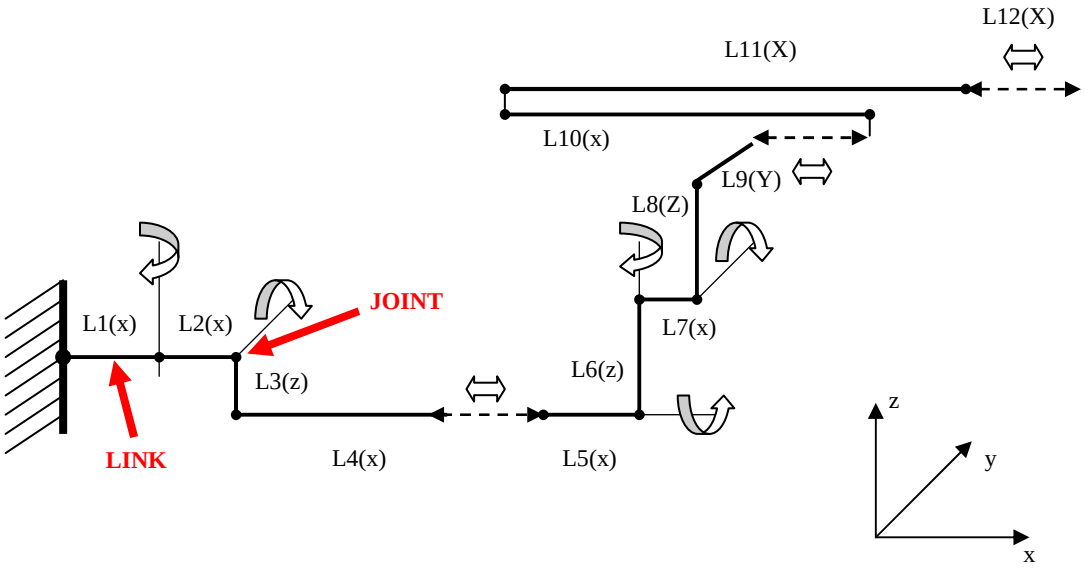


Figure 2.2: A link model

Each joint is individually maneuverable; this results in a number of actions that can be performed to position the tip of the boom, which is the drill bit. Positioning of the drill bit and – in combination – orienting the drill rod is the ultimate goal that all individual links are used for. For the boom in figure 2.1 the following actuations can be done (described using the nomenclature from RCS and the notations from the link model in figure 2.2):

- Boom swing
  - Turns L2 - L12 left or right.
- Boom lift
  - Lifts L3 - L12 up or down.
- Boom extension
  - Moves L5 - L12 frontwards or backwards.
- Feed rotation
  - Rotates L6 - L12.
- Feed swing
  - Turns L7 - L12 left or right.
- Feed tilt
  - Tilts L8 - L12 up or down.
- Feed extension
  - Moves L10 - L12 frontwards or backwards.

A boom has a sensor mounted on each joint that measures the angular, or positional, value of a joint. These measurements are used by the RCS together with the link model to determine the position of the boom and its drill bit.

2.2 The rig control system

RCS is a control system developed by Atlas Copco that is used in several of their machines. As mentioned in the introduction, it consists of a variable number of units that is connected with each other in a CAN-net. Common units in this system are:

- Application module, called *App*. There is one App for each boom on the rig and its main objective is to control the boom, by mastering the I/O modules, and keep track of its position.
- Display module, called *Disp*. This unit receives data from other units on the net and presents it on a screen. It also forwards requests from an operator panel to other units on the net to control the rig.
- I/O modules. These modules make the mechanical parts, such as hydraulic cylinders, move by operating hydraulic valves.
- Resolvers. Converts signals from the sensors, which are mounted on the joints, to angular or positional values. The values are then returned to the App and used in its calculations.

The system has a central net and one or more local nets, depending on how many booms the rig have. The local net usually contains one App and a number of I/O modules and resolvers. These nets control one boom each. The central net usually has a Disp and some I/O modules to control rig specific tasks, such as unfolding supporting jacks. A Boomer with three booms can have a setup looking like figure 2.3.

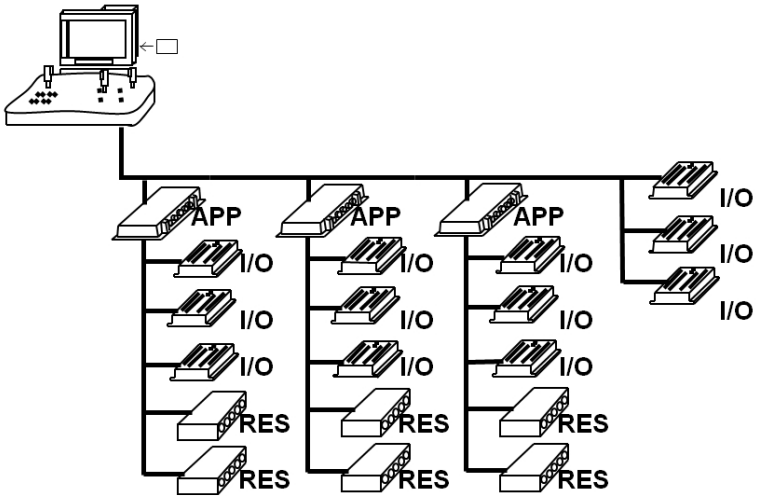


Figure 2.3: Illustration of a RCS setup

When controlling a boom the operator panel sends messages about a desired action to the Disp. The Disp forwards these messages to the targeted App which controls the I/O modules to move the boom. This triggers the sensors mounted at the joints. The sensors are connected to resolvers that convert the values and return them back to the App.

2.3 The simulator

To avoid only being able to test software for the RCS on actual hardware, Atlas Copco has developed a simulator that runs on a desktop PC.

The software for the App and the Disp can without modification be built to target the Win32 platform instead of the platform running on the hardware modules that are mounted on the rigs. But, to be able to run the App and Disp processes in a Win32 environment, the CAN bus, I/O modules and resolvers have to be simulated, which is what the simulator does. The simulator can also simulate an operator panel with joysticks and buttons that can be used to manually steer the booms. It also has an embedded TCP server which can be used by external applications to fetch information, such as signal data.

3 Implementation

This chapter describes how the main parts in the 3D visualization application were implemented. The application has one seperate thread that constantly receives new signal data from the simulator. This data is then passed to the context where the 3D models of the booms are drawn. The angles or displacements between the parts that builds up a boom are depending on the received signal values. By updating, or in other words redrawing, the context each time new signal data has arrived, the 3D models moves in the same way they should in reality.

The implementation of the application is mainly concerned with the following topics:

- TCP communication
- Threading
- 3D graphics
- Animation
- Design
- Graphical User Interface

3.1 Possible development tools

The first thing that needed to be done was to choose the development tools. This section describes the alternatives that were taken in consideration. All tools build on the C++ programming language, something that is desirable as this is the language used for all projects within Atlas Copco.

3.1.1 MFC

MFC is an abbreviation of Microsoft Foundation Classes and a C++ class library designed by Microsoft. As well as having its own container classes and encapsulate some parts of the Windows API it also includes a framework for making graphical user interfaces. MFC is integrated with MS Visual Studio, which makes it easy to create graphical Windows application using the so called drag and drop technique.

3.1.2 RFC

RFC is an abbreviation of Rig Foundation Classes and the class library designed and used at Atlas Copco. The advantage of using these classes is that they are commonly used at the company; this would make it easier for them to maintain and develop the application. If the 3D visualization should be embedded in the simulator and pull data directly from RCS it had been desirable to use RFC, since RCS is built using it. But, since the simulator provides a TCP interface, there is a way for external applications to get data without knowledge about the internals. This makes it possible to create a standalone application without RFC dependencies, which is desirable for this work.

3.1.3 Qt

Qt is an API built in C++ with the primary purpose of being a cross-platform framework for developing GUI applications. Later releases of Qt include some collection of classes, called modules, which support general software development topics, such as OpenGL, SQL, threads and network programming. The Qt SDK includes both an integrated development environment called Qt Creator and a GUI layout and form designer, called Qt Designer. Since the later releases of Qt are supported by the Microsoft Visual C++ compiler, MS Visual Studio can be used for development. Qt is also being used at Atlas Copco in some projects.

3.1.4 OpenGL
OpenGL is a cross-platform API that is used to produce 2D and 3D computer graphics. It supports a small set of primitives that can be used to describe how a model should be drawn. Since it is a platform independent interface there is no way to handle display contexts, window management and obtaining user input. This has to be handled by platform specific APIs. OpenGL also supports transformation functions, which makes it easy to rotate, translate and scale objects in the scene.

3.1.5 Conclusion
OpenGL was selected as the computer graphics API since it's easy to use and not that resource-intensive. For the GUI, threading and networking part, the Qt API was selected. One reason for this was the QtOpenGL module, a module that makes it easy to initialize and manage OpenGL display contexts using Qt. As a development environment, MS Visual Studio 2005 was chosen.

3.2 TCP Communication
To be able to communicate with the simulator and get the values for each sensor on the boom, a connection has to be established between the application and the TCP server that is hosted by the simulator. The TCP server listens on a predetermined port number and can handle one client at a time. To establish a connection and communicate with the server the application uses the Qt network module, a module that contains classes that encapsulate some networking functionality. One of these is the QTcpSocket class which is used by the application to connect to the server. This is done by calling the `connectToHost()` function, passing the desired IP address and port number as arguments. This creates a data stream between the application and the server that is used to send and receive data, which in this case always are ASCII strings. A pointer to this stream is then passed to a QTextStream object. This class provides a simple interface for reading and writing text from and to streams.

To communicate with the server, a request message that follows a specified protocol has to be sent. If the message is syntactically and grammatically correct, the server will send a response message containing the requested information. The messages are sent as strings and each request message has its corresponding response message. An example of a request message is:

```
$0137:1,0,ALL,IO_INP_BOOM_SWING*ABCD<CR><LF>
```

This message requests the value of the IO_INP_BOOM_SWING signal on the local net for all boom instances. The response message for that request could be:

```
$01B7:0001,IO_INP_BOOM_SWING,0,0,B1M,”5.120000e+003”*ABCD<CR><LF>
```

From this message the value of the IO_INP_BOOM_SWING signal can be parsed.

The server has no command buffer implemented and cannot respond to multiple requests at the same time. A strict one-request-one-response methodology is applied. This means that the client has to send a request and in return receive a response for each signal that the application needs. Using this technique for the 3D visualization application could be quite difficult since the number of needed signal values varies, depending on the number and type of booms. Fortunately, the server has an additional functionality where the client can activate a cyclic transmission of a response message, a so-called sync message. The sync

message contains all signal data used in the simulator as a single string. The string can then be parsed to extract the signal data that is needed. This functionality is used by the application.

To activate the cyclic transmission the following request message has to be sent to the simulator:

```
$013E:1,1,2*ABCD<CR><LF>
```

The first number after the colon states whether or not the server should transmit the cyclic message, zero means that it should not and one means that it should. The second number states whether or not the server should include signal data along with the sync message. The third number states the inverse frequency in which the sync message should be transmitted according to the systems frequency. For example, a frequency of two forces the server to send a sync message on every other sync period from the CAN bus. The sync period is the period in which the CAN-net checks if the connected units are alive. This is done by broadcasting a message from a master node to which all active units respond.

As mentioned, the incoming sync message contains all signals used in the simulator as one continuous string. This string is parsed to extract the signal values needed by the application. This is done by using the QString class, which handles strings in Qt and provides functions to manipulate them. The `split()` function is used to split a string into substrings wherever some given delimiters occurs. In this case the delimiters are a comma and a quotation mark. When applying this function to the following string:

```
IO_INP_FEED_SWING,1,0,ALL,"0.000000e+000
```

The function returns a list with the following strings:

```
[0] IO_INP_FEED_SWING
[1] 1
[2] 0
[3] ALL
[4] 0.000000e+000
```

The application iterates over this list to search for signals that are needed by the application. If a signal name matches one of the needed signals, the application gets the value that is four positions after the name in the list. Since the incoming messages always have the same format, the signal value will always be four positions after the signal name. The signal values are then passed to the context that draws the booms.

3.3 Threading

The application has a separate thread that constantly receives new signal data and keeps the TCP connection with the simulator alive. This is implemented by using the thread support offered by Qt. A thread is created by subclassing the QThread class and override its `run()` function. The code that shall be executed by the thread are placed inside this function. The thread is started by calling `start()`, which starts the execution of the `run()` function in a concurrent thread. The execution ends when the function returns. Figure 4.1 depicts a flowchart of the work being done by the thread in the 3D visualization application.

The 3D graphics context lives in another thread, therefore the received signal data has to be shared across threads. This is done by using Qt's signals and slots mechanism, a mechanism that is used for communication between objects. A signal, which in fact is a normal member function, can be emitted by an object when an event occurs. A slot, which also is a normal member function, can be connected to a signal and will be called when the signal is emitted. The thread that handles the TCP connection emits a signal when new data has arrived; this data is passed as an argument to the signal function. Since the signature of the connected slot, which belongs to the 3D graphics context, matches the signature of the signal, the passed argument is copied to the slot function. These data is then used to update the 3D graphics.

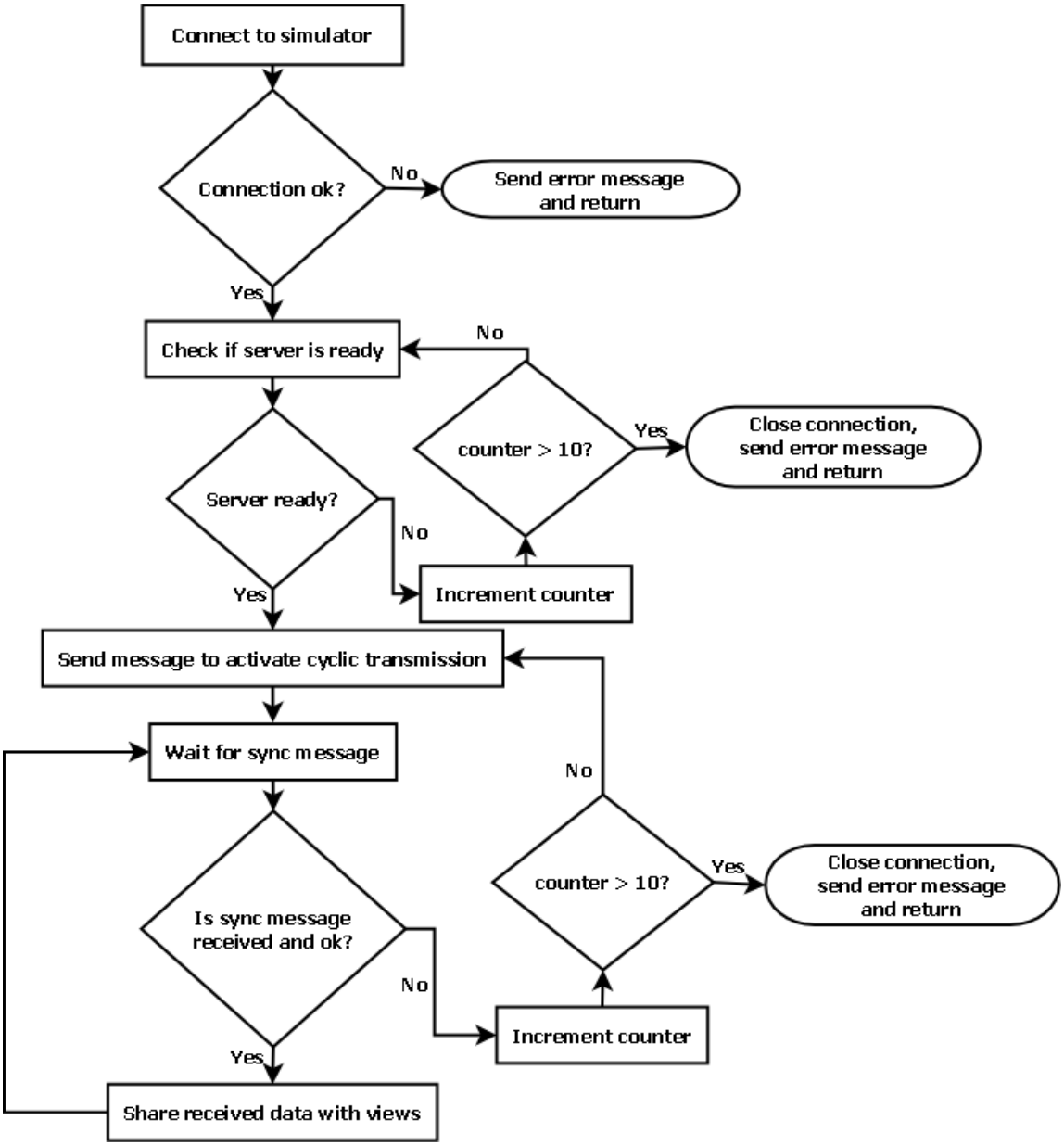


Figure 4.1: Flowchart

3.4 3D Graphics

To be able to show 3D graphics on the computer screen, a display context has to be initialized. In this application it is done by using Qt and its OpenGL module. Setting up a rendering context is done by subclassing the QGLWidget class and override the functions listed in table 4.1. Since this class is derived from the QWidget class, which is the base class for GUI objects, it can handle user interaction by catching events triggered by the keyboard and mouse.

`initializeGL()`	This function initializes the OpenGL context and is called once before the actual drawing of the scene begins. Specifying light conditions and clearing color, loading textures and initializing display lists are some of the actions performed inside this function.
`resizeGL(int w, int h)`	This function is called once after the `initilizeGL()` function and whenever the window is resized. In this function the viewport and projection transformation are set. In the application a perspective projection is used, this makes objects farther away appear smaller than closer objects with the same size.
`paintGL()`	This function is called whenever the widget has to update. In this function the actual drawing of the scene is done.

Table 4.1: Functions to setup OpenGL context with Qt

In OpenGL, a 3D model is created by concatenating multiple polygons to approximate the surface of an object. The shape of a polygon is determined by the coordinates of its vertices. To get some three-dimensionality the lightning properties for the scene, material properties for the objects and the normal for each facet have to be set up. These properties are used by OpenGL's lighting model to calculate which color each polygon should have.

The application is designed so that a boom can be modeled in two different ways. A simple way, in which each part is represented as a cuboid with six facets, and a more advanced way, in which the model is built up by multiple triangle facets. For the simple method, a class that draws a cuboid with a given width, height and length was implemented. Then, one instance of this class is created and drawn for each part of the boom. The more advanced method uses a 3D model for each part that is exported from a CAD program. The models are exported as binary STL files. These files contain a list of data that specifies the direction of the normal and the coordinates of the vertices for each triangle facet that represents the surface of the 3D model. A binary STL file has the format described in table 4.2.

Description	Type	Bytes
Header	Ascii	80
Number of triangles	Unsigned integer	4
For every triangle:		
Array of float	Array of float	4×3
Array of float	Array of float	4×3
Array of float	Array of float	4×3
Array of float	Array of float	4×3
Unsigned short	Unsigned short	2

Table 4.2: Binary STL file format

To use these models with OpenGL, the application parses the files and stores each triangle in a list. Then it iterates over the list and draws each triangle. Figure 4.2 shows both the simple and advanced 3D model of a boom.

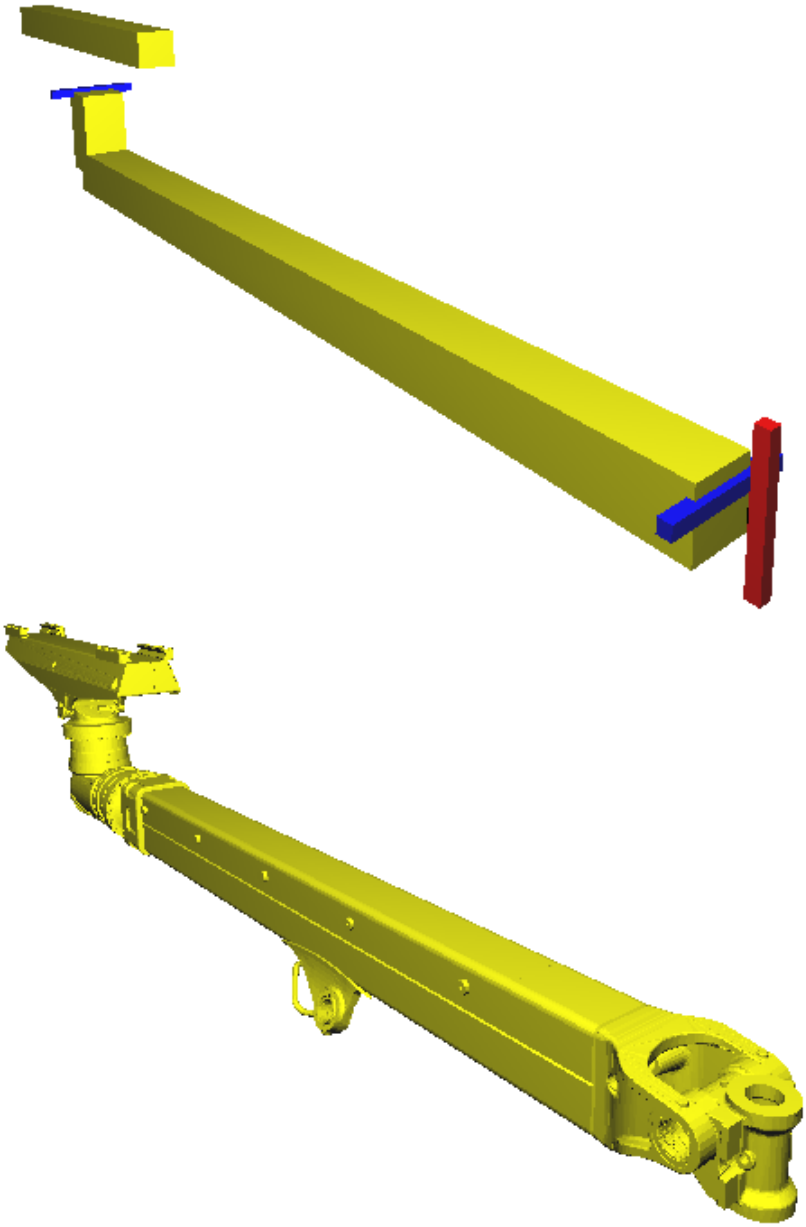


Figure 4.2: Simple and advanced 3D model of a BUT 45

The link model, described in section 2.1, is used to accurately visualize a boom. The distance between the joints are fixed and correspond to the length and direction of the intermediate link, but their position in the global coordinate system is always relative to the preceding joint. A local coordinate system is attached to each joint and the parts building the 3D model of the boom are drawn inside them. The orientation of a joint coordinate system is determined by a joint variable which, depending on the type of joint, is either an angular or a positional value. Each variable is connected to a signal value that is received from the simulator. When the application draws the model it traverses the link model, starting at the attachment point. First, a translation is done according to the distance and direction specified by the corresponding link. Then, the local coordinate system at the joint is rotated or translated according to the value of the joint variable. After this, a part of the boom is usually drawn. This sequence is done for every link in the link model. Code 4.1 shows pseudo code that depicts the actual loop that draw a boom in the applications source code.

```
for each link li in link_model
{
    translate(li.distance, li.direction)

    if li.joint_type == revolute
        rotate(li.axis, joint_variable[li.id])
    else if li.joint_type == prismatic
        translate(li.axis, joint_variable[li.id])

    draw_part(li.id)
}
```

Code 4.1: Pseudo code

The incoming signals for the revolute joints are angular values with an accuracy of one hundredth of a degree. The values for the prismatic joints are, after been multiplied with a given coefficient, in millimeters.

3.5 Animation

To animate is to create an illusion of moving objects in a scene. By rapidly showing scenes where the positions of objects change, the viewer gets the impression that the objects moves. The smoothness of an animation depends on how many frames are drawn each second. For example, a computer game that draws 60 frames per second will appear to run very smooth, while a game that only draws 10 frames per second appears as slow and irregular. How many frames that can be drawn per second basically depends on how expensive the graphics that should be drawn is and how much computer power that is available. In the application, the graphics is not that expensive, at least not when using the simple method described in section 4.3. This means that the application, on a normal desktop PC for the time being, should be able to render enough frames per second to get a smooth animation. But, since the positions of the 3D models are only changed when new data has arrived, there is no meaning to update the scene continuously. Instead, the scene is updated every time new signal data has arrived. The outcome of this is that the smoothness of the animation depends on how fast the simulator can send data. As described in section 4.1, the frequency in which new data is sent can be set in the message that activates the cyclic transmission of the sync message. If the frequency is set to one, the simulator will send a sync message every sync period, which is usually 80 ms in the simulated environment. This means that new data is sent from the simulator 12.5 times per

second. Thus, the 3D visualization will have a frame rate of 12.5 frames per second. For the purpose of the application this is enough and since the movements of the booms are quite slow the animation gets smooth. Even a lower frequency, such as every fourth sync period (3.125 frames per second) is manageable. This can be desirable since a higher frequency demands more resources which on a slower computer can lead to other services being left out.

3.6 Design

One criterion for the application was that it should be scalable. This means that it should handle different types of machines with different number and types of booms. It should also be easy to add new types. To meet this criterion, the code was designed to be as generic as possible.

Figure 4.3 shows a simplified diagram of the classes that are responsible for the final drawing of the booms.

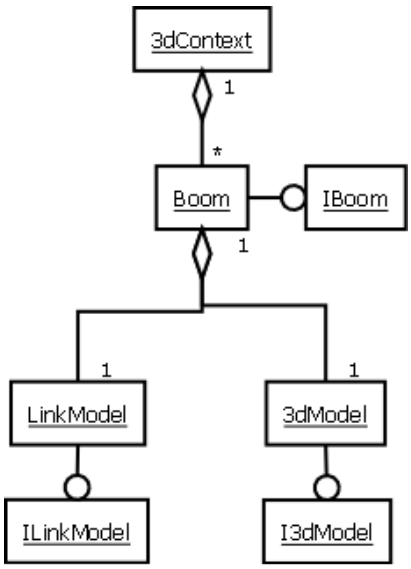


Figure 4.3: Simplified class diagram

The 3dContext initializes the OpenGL context and maintains a list of all booms, which it iterates over, when drawing the scene. The Boom class receives the incoming signals and sorts these in an array. Then, it performs the work described by the pseudo code in code 4.1. The LinkModel class represents a link model and maintains a list with information about each link. The 3dModel class contains objects that are used to draw each part of a boom.

There is one Boom class for every boom type. These classes implements the IBoom interface which specifies the functions a boom should support. By doing this, the 3dContext class can initiate a list of pointers to objects that implements the IBoom interface and, when drawing the scene, iterate over it and call the draw function for each object. Thus, the 3dContext does not need to know what kind of boom it draws. The same principle is used for individual boom types. The Boom class does not need to know what kind of link model it uses or how the 3D model looks like. The benefit of this design is that the behavior and look of a boom can easily be changed by implementing a different link and 3D model.

A configuration file specifies which machine and what types of booms that should be simulated by the simulator. The same file is used by the application to decide which booms that should be visualized. In its current state, the application supports the following boom types:

- BUT 45
- BUT 35
- BUT 32

These are the most common boom types used on the Boomer.

3.7 Graphical User Interface

The application has a simple GUI where the user can connect to the simulator, start the 3D visualization and observe signal values. It also has a log window where messages from the application are displayed. For example, if it can't read data from the simulator it will print a message in the log.

The layout of the GUI components was created by using Qt Designer and the drag and drop technique supported by it. To handle events in Qt, i.e. if a user presses a button the application should respond, the signals and slots mechanism described in section 4.2 is used. The GUI components have a number of predefined signals which are emitted when the user interacts with them. These signals can be connected to a slot function in which the responding action is implemented. A screenshot of the applications GUI is shown in figure 4.4.

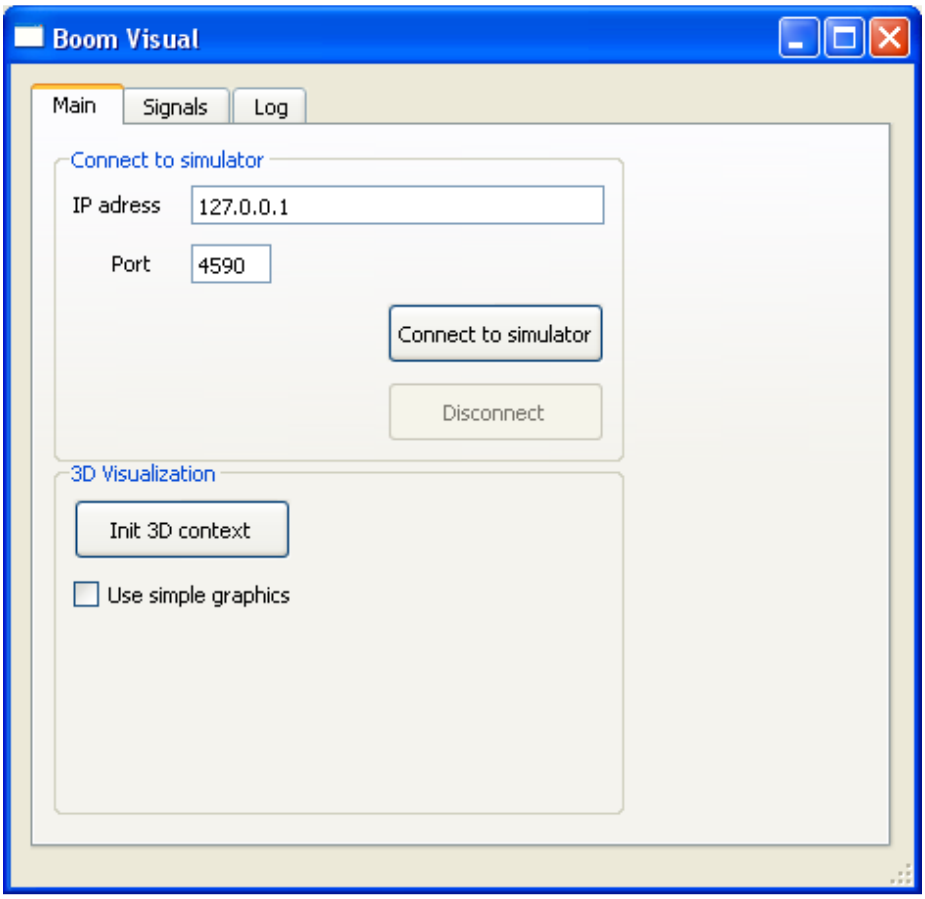


Figure 4.4: Screenshot of the GUI

In the main tab, the user can connect to the simulator by specifying the IP address of the computer in which the simulator is running. If the application runs on the same computer as the simulator, which usually will be the case, it should be the localhost address. However, if performance issues arise it can be desirable to run the application and the simulator on separate computers. The main tab also offers the user to start the 3D visualization and to choose if the simple or advanced 3D models should be used for rendering the booms.

The signal tab has a table view which can display signal names and their values. This can be used to observe how signal values change when doing some operation in the simulator. The table view has a list connected to it and the content of the list is shown in its table. The list contains signals which are updated every time new values are received from the simulator. At the moment, there is no way to specify the signals that should be observed.

The log tab contains a text component that shows messages from the application as described above. The GUI class, which contains the text component, has a slot function that takes a string as an argument and appends it to the component. By connecting signal functions, which also takes a string as an argument, to the slot function, other object are able to send messages directly to the log.

4 Results

The aim of this work was to create an application that visualizes a drill rig, with focus on its booms, using 3D graphics. It should be a standalone application and used alongside the existing simulator, with which it should communicate to retrieve signal data. These data should be used to position the booms in the visualization. The application should be able to visualize different rigs, with various numbers and types of booms.

The outcome of this work was an application that can handle three boom types and visualize their movements accurately. This is done by positioning the parts of a boom using sensor information that is received from a simulator. By reading the configuration file used by the simulator it determines which, and how many, booms that should be shown and their initial position in the scene. The application is scalable since the design of the code facilitates the addition of new types. Additional features are a GUI and a mechanism to observe signals. Figure 5.1 show a screenshot of the application.

Figure 5.1: Screenshot of the 3D visualization application

The conclusion is that the aim and requirements for this work have been fully met.

5 Discussion

Before the work begun there were speculations about how the performance of the simulator would be affected by the extra work and if a normal desktop computer running both programs simultaneously could handle it. It turned out to be some issues when using the advanced 3D models with a debug build of the application. When the models are loaded into memory, a lot of processor power is consumed; this makes the simulator unable to communicate with its processes. However, this problem did not occur when using the simple 3D models nor when running a release build. When running a release build and using the advanced 3D models the simulator can send the cyclic sync messages at the highest frequency without any problems. Older computers will have problems with the advanced 3D models and the simulator running at the same time. This could be an issue that has to be further investigated. If it is desirable to use advanced models, but reduce the demands on resources, the quality of the models could be decreased. At the moment, the models are very detailed and consist of a large amount of triangles. It should be possible to reduce the amount of triangles, but still preserve enough details for making a good representation of a boom.

The choice of development tools turned out to be successful. Qt has good support for many common programming tasks, a comprehensive documentation and useful tools. OpenGL also has a comprehensive documentation and gives good performance. The interaction between Qt and OpenGL worked from the beginning without any problems.

In its current state, the application can be used by the developers in conjunction with the simulator to see the booms position and how they behave. The following topics came up at a discussion about future development of additional features that should enhance the support brought by the application to the developers:

- Show a drill plan in front of the rig.
  A drill plan is a map of the holes that should be drilled by the rig. These plans are loaded into the RCS.

- Show collision boxes around the booms.
  Collision boxes are used in the calculations that are done by the RCS to avoid collisions between booms. This feature could help track erroneous behaviors in RCS.

- A view that can be used to observe signal values.
  A part of this mechanism is implemented, but there is no way for the user to select which signals that should be observed. The way the values are displayed also has to be improved.

- Add support for lab environment.
  It is desirable to be able to use the application in the lab environment where hardware is used in conjunction with simulation.

- Adding advanced 3D models for all boom types.
  At the moment there only exists an advanced model for one boom type, the BUT 45.

6 References

Spong, W, M, and Vidyasagar, M, *Robot dynamics and control*, John Wiley & Sons, Canada, 1989.

Stroustrup, Bjarne, *The C++ programming language*, special edition, Addison-Wesley, New Jersey, 2000.

Blanchette, Jasmine and Summerfield, Mark, *C++ GUI programming with Qt 4*, 2nd ed., Prentice-Hall, New Jersey, 2008.

Shreiner, Dave, et al, *OpenGL programming guide*, 6th ed., Addison-Wesley, New Jersey, 2008.

Buss, R, Samuel, *3-D Computer Graphics: A Mathematical Introduction with OpenGL*, Cambridge University Press, San Diego, 2003.

Parent, Rick, *Computer animation: Algoritms & Techniques*, 2nd ed., Morgan Kaufmann, Ohio, 2008

Nokia Corporation, *Creating cross-platform visualization UIs with Qt and OpenGL*, 2008.
URL (2010-04-06): http://qt.nokia.com/forms/whitepapers/reg-whitepapers-viz

Nokia Corporation, *Improving performance across platforms with Qt and multithreading*, 2008.
URL (2010-04-06): http://qt.nokia.com/forms/whitepapers/reg-whitepapers-threading-eng

Nokia Corporation, *Signals and Slots*, 2010.
URL (2010-04-06): http://doc.trolltech.com/4.6/signalsandslots.html

Burns, Marshall, *StereoLithography Interface Specification*, 3D Systems, Inc., 1989.
URL (2010-04-30): http://www.ennex.com/~fabbers/StL.asp