Examensarbete

3D Visualization Package for OpenModelica

av

Peter Arvidsson

LIU-IDA/LITH-EX-G-12/027-SE

2012-12-03
Examensarbete

3D Visualization Package for OpenModelica

av

Peter Arvidsson

LIU-IDÁ/LITH-EX-G-12/027-SE

2012-12-03

Handledare: Adeel Asghar, Olena Rogovchenko
Examinator: Prof. Peter Fritzson
Abstract

A 3D representation of a physical phenomena is often a good way to present it. It’s often easier to see an event in an animation rather than just read data tables. The goal of this thesis is to create an application that visualizes a physical simulation in Modelica. This thesis explains how the OMVisualize application was developed. To create the OMVisualize the OpenSceneGraph library was used. The OpenSceneGraph is a 3D package written in C++ that is used in many different applications from game to scientific applications.
Table of Contents

1. Introduction..........................................................................................................................1
   1.1. Background....................................................................................................................1
1.2. Problem..............................................................................................................................1
1.3. Requirements....................................................................................................................1
1.4. Glossary............................................................................................................................1

2. Method..................................................................................................................................3
   2.1. Limitations.......................................................................................................................3

3. Theory..................................................................................................................................5
   3.1. Introduction to animation ...............................................................................................5
   3.2. Background to computer animation ...............................................................................5
   3.3. 3D animation ................................................................................................................5
   3.4. Qt environment ..............................................................................................................6
   3.5. Modelica ........................................................................................................................7
   3.6. Modelica language .........................................................................................................7
   3.7. Modelica tools ................................................................................................................8
   3.8. The Modelica association ..............................................................................................8
   3.9. OpenModelica ...............................................................................................................8
   3.10. OMNotebook ................................................................................................................8
   3.11. GNU project ................................................................................................................8
   3.12. RGB color ....................................................................................................................8
   3.13. The OpenSceneGraph library (OSG) .........................................................................8
         3.13.1. OSG::Vec3 ..............................................................................................................9
         3.13.2. OSG::Triple ..........................................................................................................9
         3.13.3. OSG::Quat ..........................................................................................................10
         3.13.4. OSG::AnimationPath ..........................................................................................10
         3.13.5. Insert ..................................................................................................................10
         3.13.6. ControlPoint .......................................................................................................10
         3.13.7. shapeDrawable ...................................................................................................10
         3.13.8. setColor ..............................................................................................................11
         3.13.9. Geode ..................................................................................................................11
         3.13.10. Transform ........................................................................................................11
         3.13.11. PositionAttitudeTransform .............................................................................11
         3.13.12. setPosition ........................................................................................................11
         3.13.13. setAttitude .........................................................................................................11
         3.13.14. setTimeOffset ....................................................................................................11
   3.14. OpenGL ........................................................................................................................11
List of figures

Figure 1: Perspective projection
Figure 2: z-buffer
Figure 3: Part of OpenSceneGraph animation library
Figure 4: 3D coordinates
Figure 5: The SimpleVisual class
Figure 6: PositionRotationSizeOffset entities
Figure 7: Classes in application
Figure 8: How the windows are oriented in the QMainWindow
Figure 9: Vector
Figure 10: Application example


1. Introduction
This chapter will introduce the thesis and the motivation behind it.

1.1. Background
This report is part of the thesis work I have done at IDA at Linköpings universitet. The thesis is formulated by the institution IDA. 3D animation [see glossary] of physical phenomena is a good way to present a sequence of events that could be hard to notice, see or understand in a table or diagram. The OpenModelica [see glossary] environment doesn’t have a good 3D animation [see glossary] widget [see glossary] yet. Work has been done on other animation [see glossary] libraries so this work is concluded so that a comparison can be done. The application should be stable, fast, efficient and rely on the 3D library OpenSceneGraph [see glossary]. The application should be developed using Nokia Qt.

1.2. Problem
Here will the problems for the report be formulated.
- Is the OpenSceneGraph a library that supports good methods?
- Is the OpenSceneGraph useful for the OpenModelica?

1.3. Requirements
The purpose of this thesis is to implement a 3D visualizer with the OpenSceneGraph [see glossary] library. The 3D visualizing program will be used to show models simulated in the OpenModelica [see glossary] environment in an easy and understandable way. It should be:

- Stable
- Efficient
- Reliable

The developed application should also be able to show many simulations in one application window, single application. All the requirements come from the thesis definition made by the institution IDA. The requirements are quite natural, a visualizer must have some fundamental functions to be able to work.

1.4. Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D animation</td>
<td>§3.3</td>
</tr>
<tr>
<td>Animation path</td>
<td>§3.13.4</td>
</tr>
<tr>
<td>Modelica</td>
<td>§3.5</td>
</tr>
<tr>
<td>OMNotebook</td>
<td>§3.10</td>
</tr>
<tr>
<td>OpenGL</td>
<td>§3.14</td>
</tr>
<tr>
<td>OpenModelica</td>
<td>§3.9</td>
</tr>
<tr>
<td>OpenSceneGraph</td>
<td>§3.13</td>
</tr>
<tr>
<td>Quat</td>
<td>§3.13.3</td>
</tr>
<tr>
<td>SimpleVisual</td>
<td>§3.15</td>
</tr>
<tr>
<td>Triple</td>
<td>§3.13.2</td>
</tr>
</tbody>
</table>
2. Method
The method used to find the answer to the problems is empiricism. During the implementation of the OMVisualize program there will be theory studies. The OMVisualize will show if the OpenSceneGraph library is useful for the OpenModelica 3D animation.

2.1. Limitations
This work is only concentrated on the OpenSceneGraph [see glossary] library. No comparison to other tools should be done. The work should be done in C++ with the use of Nokia Qt development tool. The stability, efficiency and reliability are hard to measure and it will be hard to do an objective comparison.
3. Theory

3.1. Introduction to animation

Animation comes from the Latin and means “give life to”.\(^1\) This widely concept could mean everything from puppet show to the creation of Frankenstein in the Mary Shelley story *Frankenstein; or, The Modern Prometheus*. What we normally think of when we hear about animation is the painted cartoons that have been made mostly during the 1900s. Nowadays many of the cartoon movies are made by computer support. And with this computer animation ability the creation of animation have increased. Computer animation is now used in a variety of applications.

3.2. Background to computer animation

Animation with computers appeared during the 1960s. In 1963 Ivan Sutherland did the first research in animation and computer graphics.\(^1\) The early animations were only animated in two dimensions but now animation in three dimensions is widely spread. 3D animation is used in applications for medical use, movies, science etc. It has its advantages in visualizing complicated sequences in an easy way. A computer animation is built up in the same way as a traditional painted animation by frames. The animation shows usually 24 or 30 frames per second\(^1\), if the frame rate is lower humans will detect the movie as jagged. The difference between a computer animated and a traditionally painted movie is that you as creator of the computer animation just have to choose which objects that should be in which place, in what time etc the computer software will handle the painting of the objects and calculate the path the object will move along between user defined coordinates in the animation. Computer software calculates shadows and light after users wish and depending on where user has put light sources and the intensity of these. The computer software also takes care of which parts of the animation should be shown on screen i.e. what sides of the object is actually seen by the user, some parts could be hidden by another part.

3.3. 3D animation

Computer animation is used in making movies, games and to show all sorts of graphics. 3D graphics are representing objects in the room opposite to 2D graphics that is flat. The part of the room that is shown on computer screen is called the frustum. The frustum is defined as: \(\text{Frustum}(X_{\text{left}}, X_{\text{right}}, Y_{\text{bottom}}, Y_{\text{top}}, Z_{\text{near}}, Z_{\text{far}})\). This means that users define left-right, up-down limits and how far and close he can see objects, i.e. everything between these limits will be painted on screen.
When many objects or if a single object have sides that hide another side of object. All 3D objects have sides that hide other sides i.e. you can’t see the backside of a football. To handle this phenomena the z-buffer is used when doing computer animations. The z-buffer remembers the depth of a single object and from this depth it is possible to know if an object is hiding another or not. A cube is an object but could also be seen as it is built of many small objects i.e. one pixel could be seen as one object so the pixels on the backside is hidden behind the pixels on the front side.

In the figure2, z-buffer the cone is covering the sphere i.e. the cone is closer to viewer and as in real world you can’t see through a solid object and see what is behind it. The correct behavior of this covering is calculated with use of z-buffer.

3.4. Qt environment
The Qt environment is a development environment developed by Nokia. It is platform independent and is used to develop programs in C++. It supports the C++ compiler MinGw that is used to build the OMVisualize application. The Qt has functionalities that make programming easier, it has support for many widgets for example buttons and sliders. In the development of the OMVisualize many of the specially Qt functions and variables are used because they are most of the times better than their C++ standard equivalent.
3.5. Modelica

“Modelica is a non-proprietary, object-oriented, equation based language to conveniently model complex physical systems containing, e.g., mechanical, electrical, electronic, hydraulic, thermal, control, electric power or process-oriented subcomponents.” There are four important features in Modelica. These are:

- It is equation based
- It has multi domain modeling capability
- It is object oriented
- It has a strong software component

The first Modelica version was released in 1997. From year 2000 the Modelica is managed by the Modelica association.

3.6. Modelica language

The Modelica language is an object oriented language that is used to describe physical systems by text. Algebraic, differential and discrete equations are used to build the model. The code in below is Modelica code that describes the physics of a pendulum.

```
1. class MyPendulum3D "Planar 3D Pendulum"
2. import Modelica.SimpleVisual;
3. constant Real PI=3.141592653589793;
4. parameter Real m=1, g=9.81, L=5.;
5. Real F;
6. Real x(start=5.),y(start=0.);
7. Real vx,vy;
8. // Visualization
9. SimpleVisual.PositionRotationSize sphere "color=red;shape=ball;"
10. SimpleVisual.PositionSize fixture "color=black;shape=cube;"
11. SimpleVisual.PositionRotationSizeOffset thread "color=black;shape=cube;"
12. equation
13. fixture.size = {0.5, 0.1, 0.5};
14. fixture.frame_a = {0., 0., 0.};
15. sphere.size = {1., 1., 1.};
16. sphere.frame_a = (x, y, 0.);
17. sphere.frame_b = (0., 0., 0.);
18. thread.size[1]=0.05;
20. thread.size[3]=0.05;
21. thread.frame_a[1]=x;
22. thread.frame_a[2]=y;
23. thread.frame_a[3]=0.0;
24. thread.frame_b = {0., 0., 0.};
25. thread.offset[1]=0.0;
26. thread.offset[2]=L/2;
27. thread.offset[3]=0.0;
28. m*der(vx)=-(x/L)*F;
29. m*der(vy)=-(y/L)*F-m*g;
30. der(x)=vx;
31. der(y)=vy;
32. x^2+y^2=L^2;
33. end MyPendulum3D;
```

Row number one defines the new class MyPendulum3D. Row number two imports the SimpleVisual [see glossary] library to the class, this is required because the example creates a model that could be visualized with applications that need this library. Rows three to seven creates variables and constants. The code on rows nine to eleven creates the objects that will be animated in the visualizer. The code on row 13 to 23 defines size and position for the objects. Row 28 to 32 is the equations that simulate the model. After that the class is closed. The Modelica language is good and easy to use when it comes to writing equations, this is done in a natural way and is closer to the process of solving equations by hand then many other programming languages.
3.7. Modelica tools
A development environment is needed to edit the Modelica language. Many development environments are available, both commercial and free of charge. One of the free simulation environments is OpenModelica [see glossary]. There is an environment called DrModelica that could be used for educational applications. It contains examples and questions in an OMNotebook [see glossary] environment. A number of text editors that are adapted to Modelica are also available as well as other tools that combined with a development environment could make a developers life easier.\textsuperscript{5}

3.8. The Modelica association
The Modelica association is an organization that manages, develops and promotes the Modelica language. The association owns incorporeal rights related to Modelica.\textsuperscript{6}

3.9. OpenModelica
OpenModelica was created to support Modelica simulation based on free software. The consortium was founded in 2007. The OpenModelica environment contains OMNotebook [see glossary], OMSHELL, DrControl, OMEdit and some more tools that together constitute the OpenModelica environment.\textsuperscript{7}

3.10. OMNotebook
OMNotebook is based on literal programming.\textsuperscript{8} It is a part of the OpenModelica environment. OpenModelica models can be constructed, simulated and plotted in the OMNotebook.

3.11. GNU project
The GNU project was started by Richard Stallman in 1984. The goal of the project was to make a UNIX similar operating system based on free software. The GNU project also created the GNU general public license. The purpose of this license was that everybody can study and develop the source code freely.\textsuperscript{9}

3.12. RGB color
The RGB model for color representation is a model were red, green and blue color are added together to build up a wide range of colors. The RGB color model is used in a wide range of products, computer, TV screens, digital cameras and scanners are some examples. One bad thing with the RGB color representation is that it is device dependent i.e. various screens and even same screen show colors differently over time. To create a color with the RGB method a vector of numbers from 0 to 1 is created. (0, 1, 0) creates a color that has no red full green and no blue component to the resulting color, it will show a pure blue color.\textsuperscript{10}

3.13. The OpenSceneGraph library (OSG)
This work is concentrated on using the OpenSceneGraph library to do the 3D animation. The tool is written in C++ and supports many platforms, for example Windows, Linux and Solaris. The development started 1998 and since then updates have been released regularly. It supports OpenGL [see glossary] and makes things easier for the developer. The developer does not have to code the low level graphics calls. OpenSceneGraph was originally created on IRIX and then ported to Linux, then to Windows, then FreeBSD, Mac OSX, Solaris, HP-UX, AIX and Playstation2.\textsuperscript{11} OpenSceneGraph use the scene graph technique to do the animating. Scene graph technique is used by many modern tools to make animations. The scene graph is built up in a tree like structure, a good thing with this is the possibility to affect a parent and automatically propagate this affection to its children. A compound object is easily built up and affected by this structure.\textsuperscript{12} OSGViewer is used to integrate OpenSceneGraph with for example Qt. The following OSG-functions are the ones that are used in the OMVisualize application.
In the following paragraphs I will briefly present the most important OpenSceneGraph functions and variables used in the OMVisualize.\textsuperscript{13}

\subsection*{3.13.1. OSG::Vec3}
OpenSceneGraph triple [see glossary] that is used for vectors. In the visualize application it is used to store positions and in the case a quat [see glossary] it is used to hold axis. The vec3d stores doubles and the vec3f handles float numbers.\textsuperscript{12}

\subsection*{3.13.2. OSG::Triple}
An OpenSceneGraph triple is a vector containing three numbers, this is very useful in 3D animation. With three values it is possible to describe a point anywhere in space. Any point P could be exactly placed by describing its position by height (Z), deep (X) and width (Y) from a point in room that have coordinates (0, 0, 0).\textsuperscript{12}
3.13.3. **OSG::Quat**

The OpenSceneGraph quat is a datatype that is used to rotate nodes or groups of nodes. It is used with an angle for every axis i.e.

```cpp
gosg::Quat{
    angleX, osg::Vec3d(1.0, 0.0, 0.0),
    angleY, osg::Vec3d(0.0, 1.0, 0.0),
    angleZ, osg::Vec3d(0.0, 0.0, 1.0)
};
```

The angles are expressed in radians. And the Vec3d: s is defining which axis the angle should be set on. The example code will let the quat rotate angleX around the x-axis, angleY around the y-axis and angleZ around the z-axis.\(^1\)

3.13.4. **OSG::AnimationPath**

The `AnimationPath` encapsulates a time varying transformation pathway.\(^2\) It is used to update model and camera position over time.

3.13.5. **Insert**

The `animationPath->insert` is used to insert time and position into the `animationPath`.

```cpp
void osg::AnimationPath::insert(double time, const ControlPoint &controlPoint)
```

The insert function adds a time and `controlPoint` to the `animationPath` every time it is called so to fill the whole animation with data the insert is called as many times as there is data. In the visualize application it is filled as many times that there is simulated positions.\(^2\)

3.13.6. **ControlPoint**

An OpenSceneGraph `controlPoint` is a class member that holds either only position, position and rotation or position, rotation and scale.\(^3\) In this application is the default constructor without arguments used. The rotation and position is then set by the `setRotation` and `setPosition` functions. The rotation is always calculated from the original rotation i.e. how much the object is rotated when created.

3.13.7. **shapeDrawable**

The function `shapeDrawable` is an easy way of creating objects with a shape of a basic geometry. This creation of a `shapeDrawable` is a good way to create simple animations as those that will be created in the OMVisualize. If more complex animations and objects should be created this method is not the best and most efficient.\(^2\)
3.13.8. **setColor**
void osg::ShapeDrawable::setColor (const Vec4 & color) is a good way to set a color for an object. The function defines the color with the RGB color definition.\(^\text{12}\)

3.13.9. **Geode**
A geode is a node that holds geometry. It is a leaf node to a sceneGraph that holds drawable objects.\(^\text{12}\) This means that all shapes that will be animated must be attached to a geode.

3.13.10. **Transform**
The transform is a group node that children are transformed to. The transform is used to position objects in a scene. If positionAttitudeTransform is connected to a transform transformations could be done to objects.\(^\text{12}\)

3.13.11. **PositionAttitudeTransform**
The positionAttitudeTransform or PAT is used to manipulate objects in the animation. All objects are pushed through the PAT during time. Every time an object moves it has to be translated to the new position. Rotation is also a transformation so rotation and position is handled in similar ways.\(^\text{12}\)

3.13.12. **setPosition**
The setPosition could be used in a very easy way to create a moving object. It is possible to do a simple loop where setPosition and frame drawing is called for every loop. This will cause the object to move.\(^\text{12}\)

```cpp
While(moving)
{
    setPosition(osg::Vec3d(x, y, z));
    drawFrame;
    change x, y, z
}
```

3.13.13. **setAttitude**
This is instead of moving the object in the room causing it to rotate. It could be used in the same way as the setPosition.\(^\text{12}\)

3.13.14. **setTimeOffset**
This function is used by the time slider and button to go back and forward in the animation. It is used on the animationPath as follow:

```cpp
this->mCallBack[r].rAniCallback->setTimeOffset(0);
```

The code above will set the animationPath time to 0 i.e. the animation will start from beginning.\(^\text{12}\)

3.14. **OpenGL**
Open graphics library normally called **OpenGL** was created by Silicon graphic in 1992 and is a platform that is used to create 2D and 3D animations with. It standardized how to use hardware when programming graphics. It is now managed by the Khronos group. **OpenGL** is a standard for how to write an application that shows 3D or 2D animations.\(^\text{14}\)

3.15. **SimpleVisual library**
The simple visual library holds a number of classes that can be connected to variables in a simulation model.
The different classes have more or less entities connected to it. The \textit{PositionRotationSizeOffset} is the most complicated class and it holds the highest number of entities of the classes.

\textbf{Figure 6:} PositionRotationSizeOffset entities

The position class only has the \texttt{frame\_a} entity. The \textit{positionRotation} class have \texttt{frame\_a} and \texttt{frame\_b}. These two are not that interesting in visualize because the animated object have no size. And no size means that the object would not be seen. The \textit{positionSize} class holds \texttt{frame\_a} and object size, this could be used in simple visualizations as the BouncingBall3D. PositionRotationSize class could be used in simulations and visualizations where one or more objects have (0,0,0) as rotation point. The \textit{PositionRotationSizeOffset} class holds all the variables that an object can have. The position is the position that the object has and is controlled by the \texttt{frame\_a} variable. Rotation can be used if an object is rotating during the animation and this is controlled by the \texttt{frame\_b} variable. The vector that can be calculated between \texttt{frame\_a} and \texttt{frame\_b} is used to set the direction of the objects rotation. Size is how big the object is and is controlled by the size variable. The offset can be used if the object is moved from the original position. Offset can also be used to rotate around another position than centre of the animation, this is controlled by the offset variable.\textsuperscript{15} The class also holds the shapes cube, sphere and cylinder.
4. Implementation and design

This chapter will focus on how the OMVisualize application was created and why it was made in a special way. From downloading the OpenSceneGraph library, adapting it to the Qt environment, creating the file readers, get a working application and testing it.

4.1. Qt Main Window framework

![Diagram of QMainWindow](http://qt-project.org/doc/qt-4.8/qmainwindow.html#details)

Figure 8: How the windows are oriented in the QMainWindow.

The OMVisualize is made as a QMainWindow. The QMainWindow gives a lot of support that OMVisualize use. In figure 8 you can see how the main window in Qt is built. It is possible to use some of the areas or all of the areas. In the OMVisualize is the menu bar the central widget. The status bar is also used.

4.2. Classes in the application

OMVisualize contain three classes, these are the visualizeMainWindow, the visualizeWindowContainer and the visualizeWindow. The structure with three classes is natural. With these classes the OMVisualize was divided in a good and functional way.
4.2.1. **VisualizeMainWindow**

This class inherits the QMainWindow. The toolbar and the dock widget are not used in this application. In the OMVisualize application the menu bar is used. The central widget contains the `visualizeWindowContainer`.

4.2.2. **VisualizeWindowContainer**

This class is a QMdiArea. QMdiArea holds instances of QSubWindows. QSubwindow in the OMVisualize case is a QWidget. The QMdiArea is often used as central widget in a QMainWindow. This is how it’s used in this application.

4.2.3. **VisualizeWindow**

The `visualizeWindow` class is the class that is used to store the animation data, sizes and colors of objects. It also holds all the application window data, as buttons and sliders. The class inherits from the QtWidget class.

4.3. **The OpenSceneGraph window**

The context was limited to OpenSceneGraph so no time was spent on choosing the package for the 3D implementation. This limitation was made by the project specifications. The work doesn’t connect that much with the OpenModelica code. The OMVisualize only needs to be able to read the output files from the simulation and being reachable from OMNotebook.

Some problems appeared during the process to get the OpenSceneGraph library compiled and working with the Qt environment. The problems were about how and with what tool the OpenSceneGraph library should be compiled. After that the problems was solved the library and the Qt environment worked well together. Some OpenSceneGraph examples were made to get familiar with the package. It is easy to get started with it because OpenSceneGraph automatically creates light and has several predefined shapes ready to be used.

The OpenSceneGraph handles object movement in a way that is easy to use, the OpenSceneGraph use `AnimationPath` that is a good support when creating an animation. The AnimationPath class has a member called `controlPoint` that hold the position for every animated object. The controlPoint can also hold
rotation. The rotation are used when the SimpleVisual.PositionRotationSizeOffset or SimpleVisual.PositionRotationSize are used for the OpenModelica model being simulated.

```cpp
osg::AnimationPath::ControlPoint::ControlPoint(const osg::Vec3d & position, const osg::Quat & rotation)
```

In OMVisualize is an empty controlPoint created that become filled with data during program runtime.

```cpp
osg::AnimationPath::ControlPoint c0;
... c0.setRotation(osg::Quat(angleX, osg::Vec3d(1.0, 0.0, 0.0),
angleY, osg::Vec3d(0.0, 1.0, 0.0),
angleZ, osg::Vec3d(0.0, 0.0, 1.0)));
... c0.setPosition(osg::Vec3(this->mAnimVectors[i].vect[j+1],
this->mAnimVectors[i].vect[j+3],
this->mAnimVectors[i].vect[j+2]));
```

Filling the AnimationPath is done by:

```cpp
void osg::AnimationPath::insert(double time, const ControlPoint & controlPoint)
```

This means that at a specific time the object will be at place controlPoint that contain a vector with x, y, z variable and rotation in some cases. The OpenSceneGraph then calculates speed and direction for all objects with the help of AnimationPath data.

### 4.4. Supported file formats

OMVisualize supports the three file formats that are possible for OpenModelica to deliver simulation data in. The supported files are AutoCad plot file (plt), Comma separated vector file (csv) and Matlab file (mat).

#### 4.4.1. plt

The plt file is a text file that is easy to read directly as a text file. It is built so that a variable is followed by its values.

```plaintext
... DataSet: time
  0, 0
  0.02, 0.02
  0.04, 0.04
  0.06, 0.06
... DataSet: der(der(y))
  0, -9.81000000000001
  0.02, -9.809991690622734
  0.04, -9.809910545236591
...```

#### 4.4.2. csv

The csv file is a delimited file using comma for separating the data. It is more complicated to read than the plt file but it is still readable in a text editor. A csv file looks like this:

```plaintext
... "time","vx","vy",
0,0,0,
0.02,-0.000119310084374849,-0.1961999515172131,
...```

The first row define the names of the variables and could be used to find out what column in the following file it will have its data in. To find the vx variable data you see that the vx is in column number two, this gives you that all data in column number two are the vx data.
4.4.3. mat
The mat files are binary and impossible for humans to read in a text editor and a special program must be used to get the data.

4.4.4. visualize
This file is created by OpenModelica to communicate to OMVisualize what color and shape the animated object should have and what SimpleVisual mode it has.

In this example the first word Sphere1 is the name of the Modelica variable. The color defines the color and the shape the shape. The last word, SimpleVisual.PositionSize defines how many variables i.e. how the object should be animated. In this example only position and size of the object is defined and the file reader should only read variables defining this.

```
... SimpleVisual.PositionRotationSize sphere "color=red;shape=sphere;":SimpleVisual.PositionSize
```

The model above correspond with the visualize file below. Here you can see how the Modelica variable fixture in the visualize file become connected by the shape cube, color black and gets the SimpleVisual mode PositionSize. All the variables are connected in the same way. When the OMVisualize reads the visualize file the application gets the needed data to know how to create the animation objects.

```
SimpleVisual.PositionRotationSize sphere "color=red;shape=sphere;":SimpleVisual.PositionSize
```

4.5. File readers
The simulation file could be in three different formats so the file extension has to be read. To choose which reader the OMVisualize should use the program compare the suffix for the file name the user want to animate. This is made three times, one for every approved file type. The code to check if the file is a mat file is shown below. The code for the other two file types are the same except the comparison string.

```
if (simulationFileInfo.suffix().compare("mat") == 0)
```

After some research about how the readers should be implemented, an issue became obvious. It was impossible to get the name of the Modelica simpleVisual objects to the OMVisualize from the simulation.
file. The file readers need the `simpleVisual` name because the readers sort and extract all objects by this name. The solution was to create a file that contains the object name, color, shape and the `simpleVisual` type. These data are used to find all objects in the simulation file. The file containing the object data is the visualize file. To extract the wanted data from the visualize file it is important that the visualize file have correct format, the OMVisualize reader extracts the data from the file depending on which place in the file it has.

```cpp
while (!textStream->atEnd())
{
    currentLine = textStream->readLine();
    currentLine.remove('"');
    currentLine.remove("color=");
    currentLine.remove("shape=");
    currentLine.remove("SimpleVisual.");
    tempList = currentLine.split(";");
    values = tempList[0].split(",");
    tempList.removeAt(0);
    values.append(tempList);
    aObj.objectName = values.at(0);
    aObj.color = values.at(1);
    aObj.shape = values.at(2);
    aObj.sVisual = values.at(3);
    this->mObjVector.push_back(aObj); //Putting everything together to the object
}
```

In the visualize file one row is one object, this is used to get how many objects that should be created. A vector of structs that contain all data from visualize file are created. The struct look like:

```cpp
struct obj{
    QString objectName;
    QString shape;
    QString color;
    QString sVisual;
    float objSizeX;
    float objSizeY;
    float objSizeZ;
    float offsetX;
    float offsetY;
    float offsetZ;
};
```

Every object that should be animated is represented by a struct like this. The OMVisualize organize all the structs in a vector. It will therefore create a vector that has the size of as many objects that should be animated.

```cpp
std::vector<obj> mObjVector;
```

This structure is chosen because the standard vector has functions that are useful and well known. The sizes and offsets are extracted from simulation file in the appropriate simulation file reader. This way the OMVisualize get a good and structured container to store object data in. The container is reachable wherever it’s needed in the code.

The word in visualize file `SimpleVisual.PositionSize` will get the reader to only read the position `Frame_a` and size variables from file. The `SimpleVisual.PositionRotationSize` will set the application to also read the `Frame_b` variables to be able to rotate. If the `SimpleVisual.PositionRotationSizeOffset` is used even the offset variables will be read.
The positions and times are extracted from the simulation file that OMNotebook provide after simulation. To get the size in x variable, length of an object the reader takes the name of the object adds .size[1] and searches the simulation file for this name. All the variables that OMVisualize need are added to the variablesList QStringList. The digit inside the brackets corresponds as: 1 = x, 2 = y, 3 = z. This comparison is made in the same way for all three file formats.

```cpp
//Filling up QStringList with the interesting variable names
variablesList.append("time");
variablesList.append(".frame_a[1]");
variablesList.append(".frame_a[2]");
variablesList.append(".frame_a[3]");
variablesList.append(".frame_b[1]");
variablesList.append(".frame_b[2]");
variablesList.append(".frame_b[3]");
variablesList.append(".size[1]");
variablesList.append(".size[2]");
variablesList.append(".size[3]");
variablesList.append(".offset[1]");
variablesList.append(".offset[2]");
variablesList.append(".offset[3]");
```

In the application the objects sizes and offsets are extracted and saved in the struct obj, the same struct as the visualize file data are saved in. All strings are not used for all models, depending on what simpleVisual type the model has. In OMVisualize all the strings above will be created even if they are not needed. The program will find out which variables that are needed and witch not being used. Variables that not are used are being set to 0. This solution was selected to always get a special data on a fixed place and to prevent undefined variables.

The plt file is the easiest file format of the three simulation files to read for humans so it was the first to be implemented i.e. it was easy because the file is simple to read for humans so it can be compared with the data the animation get. The reader takes line after line from the file and extract wanted data from it with help from the variablesList to know what data that are needed.

```cpp
... if(currentVariable == variablesList.at(1))
    xVector.push_back(QString(values[1]).toDouble());
... 
```

The data are put in different vectors depending on what data that are extracted. The vectors are:

```plaintext
timeVector
xVector
yVector
zVector
xVectorB
yVectorB
zVectorB
```

When all data are collected these vectors are put in two vectors, mAnimVectorsB and mAnimVectors. The mAnimVectorsB would contain the time vector and the three B vectors if SimpleVisual.PositionRotationSize or SimpleVisual.PositionRotationSizeOffset are used, otherwise it will contain 0. The mAnimVectors will have timeVector on position 0, xVector on position 1, yVector on position 2 and zVector on position 3. This pattern will repeat as far as it needs to be.

```plaintext
mAnimVectors[time, x, y, z, time, x, y, z,...]
```
These two vectors are pushed to every object that should be animated. After some work on the reader it worked but some optimization and simplification was needed. When the reader has done the work there would be as many containers that there are objects to be animated containing:

mObjVector ->ObjectName;
   ->shape;
   ->color;
   ->sVisual;
   ->objSizeX;
   ->objSizeY;
   ->objSizeZ;
   ->offsetX;
   ->offsetY;
   ->offsetZ;

mAnimVectors ->time
   ->ax
   ->ay
   ->az

mAnimVectorsB ->time
   ->bx
   ->by
   ->bz

When this first reader worked and delivered correct data to the animation it was possible to see how the animation worked with actual simulation data. Of course there were some connection problem but after some changes in the formatting of the vectors that are filled with the simulation file data the animation worked. The readPltFile is the function that does the reading of plt files.

Next file reader that was made was the readCsvFile. This reader can read csv files. Csv files are also possible to read for humans so it was possible to know which data that should be read and therefore possible to see if the data that reached the animation was the correct data. This time the animation worked with the imported data because most of the problems related to file reading were taken care of with the plt file. This reader delivers an object that looks the same and contains the same data as the plt reader does.

The last file format that needed to be implemented was the mat file. The function readMatFile was made to do the reading. The mat file is impossible to read for humans and this makes it harder to know if the animation behavior corresponds to the behavior described by the simulation results. This problem was solved by making a simulation that was saved as both plt and mat format. This way it was possible to know if the animation behavior was correct. In this application an already made mat reader was used so some of the work was already done, the work was to know how to get the correct data from the mat reader and sort them in a correct manner. I used the read_matlab4 which are delivered in the OpenModelica trunk. The mat reader deliver data in the same format as the plt and csv reader does.

4.6. Rotation

To get the rotation of an object to work some computations have to be done. In a simulation that result in rotation of objects the simulation will provide a frame_a and frame_b vector in the simulation file. The objects that should rotate must have SimpleVisual mode PositionRotation, PositionRotationSize or PositionRotationSizeOffset to be able to support frame_b, rotation. OpenSceneGraph creates rotation with an angle so the vector from frame_a to frame_b had to be created. This vector between frame_a and frame_b are then used to calculate the angle between this vector and in this case the Y-axis. This is done by:

```
osg::Vec3f A = osg::Vec3f(this->mAnimVectors[i].vect[j+1],
                      this->mAnimVectors[i].vect[j+2], this->mAnimVectors[i].vect[j+3]);
osg::Vec3f B = osg::Vec3f(this->mAnimVectorsB[i].vect[j+1],
```
osg::Vec3f A is the point that is described by frame_a, in cases that not have any offset it’s (0, 0, 0), i.e. the point that will be the centre in the rotation. The vec3f B is the point that frame_b describes, i.e. the point that the vector points to.

\[
\begin{align*}
C &= \mathbf{A} - \mathbf{B} \\
C &= C / C.\text{length}(); // Normalize the resulting vector \\
\text{angleX} &= \text{std::atan2(C.z(), C.y())}; \\
\text{angleY} &= \text{std::atan2(C.x(), C.y())}; \\
// \text{angleZ} // Is not implemented \\
c0.\text{setRotation}(\text{osg::Quat(\text{angleX}, \text{osg::Vec3d(1.0, 0.0, 0.0)},} \\
&\text{\quad angleY, \text{osg::Vec3d(0.0, 1.0, 0.0)},} \\
&\text{\quad angleZ, \text{osg::Vec3d(0.0, 0.0, 1.0))});
\end{align*}
\]

 osg::Vec3f A is the point that is described by frame_a, in cases that not have any offset it’s (0, 0, 0), i.e. the point that will be the centre in the rotation. The vec3f B is the point that frame_b describes, i.e. the point that the vector points to.

\[
(bX, bY, bZ) \\
C \\
(aX, aY, aZ)
\]

Figure 9: Vector

The vector C is the vector between point A, (aX, aY, aZ), and point B, (bX, bY, bZ), this vector must be normalized to be used in the atan2 function. The atan2 function is an arctangent function that takes two arguments.

\[
\text{float atan2(float y, float x);} \;
\]

It returns the arc tangent for y/x in radians and determines the correct quadrant for the angle with the sign of the variables. In OpenSceneGraph the angle is set by \textit{setRotation} on the AnimationPath::ControlPoint. The \textit{setRotation} takes six arguments i.e. three angles and on which axis the angle would be implemented on.

4.7. Interactions

To make the animation more usable some interactions have to be applied. The interactions OMVisualize have implemented are the play/ pause, time slider, zoom slider, speed slider, fit to window and Speed 1. The interaction possibilities are implemented in different ways. A stop button was implemented in early implementations but was later removed. A stop function is meaningless, if user want to stop animation the pause could be pressed.

4.7.1. Play/ Pause button

The first one I will explain is the Play/ Pause button. This button is the same, it changes behavior depending on if the animation is paused or played. This button affects the AnimationPath with the following function if it is in Pause mode.

\[
\text{float atan2(float y, float x);} \;
\]
The function will change the buttons mode to Paused and change the text on the button to PAUSE. It will also affect the animationPath for all the objects and sets it to run with the setPause(FALSE) function. When in Play mode the opposite of the Pause mode will happen.

### 4.7.2. Restart button

The Restart button will set the animation to start position. When pushing it all animationPaths will reset, set the animation to paused, move sliders to start position and set all the text boxes to start number. As well as setting the text on buttons to start text.

```cpp
void VisualizeWindow::restartSlot()
{
    for (unsigned int r = 0; r < this->mObjVector.size(); r++)
    {
        this->mCallBack[r].rAniCallback->AnimationPathCallback::reset(); //Reset animation
        this->mCallBack[r].rAniCallback->AnimationPathCallback::setPause(FALSE); //Start animation
        this->mpSliderTimer->start(); //Slider set to start
        this->mProgressTime = 0; //No progress
        this->mpProgressBar->setValue(0);
        this->mpNowTime->setNum(0); //Where animation is, time = 0 on restart
        this->mCallBack[r].rAniCallback->setTimeOffset(0); //Start animation from beginning
        this->mpPlayButton->setText("PAUSE");
        this->mPausePressed = FALSE;
    }
}
```

### 4.7.3. Time slider

The time slider functionality was a bit harder to implement. First of all the slider uses ticks i.e. fixed steps that it could be dragged between. This behavior was not wanted so normalization was needed so that the ticks become so small that they can’t be noticed. Another problem was to solve that when moving slider the animation jagged. This problem was solved by setting animation to pause every time slider is moved. The slider uses the OpenSceneGraph function setTimeOffset this function is used to go to a specific time in the animation. Every time the slider is moved the animation restarts and goes to the time the slider is indicating. In this slot it is important to remember that the slider is normalized with 1000.

```cpp
void VisualizeWindow::slideChangeSlot(int slideValue)
{
    this->slider = double(slideValue); //Set value
    for (unsigned int r = 0; r < this->mObjVector.size(); r++)
    {
        if (this->mPausePressed==FALSE) //If user change slider without have pressed pause before
        {
            for (unsigned int r = 0; r < this->mObjVector.size(); r++)
            {
                this->mCallBack[r].rAniCallback
```
4.7.4. Zoom slider

Zoom functionality is possible with the OpenGL functionality by clicking and dragging with mouse in the animation window. The OMVisualize also have a slider that could be used for zoom. The zoom slider affects the home position of the camera. It is only possible to zoom in Z-axis i.e. towards or away. It is possible to make sliders that move the camera in any direction but I couldn’t see the advantage of this, it is always possible to use the mouse in the animation window to get free movement of camera. The zoom slider has the same normalization behavior as the time slider.

4.7.5. Speed slider

To change speed is a functionality that’s good to have. If a simulation is very long, things happen really slow or fast speed changing is usable. The OpenSceneGraph function setTimeMultiplier(double) is used to get this functionality. This function is easy to use, just give it a double and it will multiply the current running speed with this number. The speed slider has the same normalization behavior as the time slider. I have chosen to set max speed to 100 times original speed and minimum speed to 0,1 times original speed.

4.7.6. Fit to window and Speed 1

The buttons “Fit to window” and “Speed 1” are used to set speed and zoom to default values.

4.8. Menu functions

In the menus in the main window the user can set some options. The choices are: Close the program, Tabbed view or not.
4.9. Testing
The application was tested on the Modelica models BouncingBall3D and Pendulum3D. The BouncingBall3D is a sphere that is dropped from a height and then bounces against a floor surface and finally stops. It contains only one object and will create a Visualize file that will look like the following:

```
sphere,"color=red;shape=sphere;":SimpleVisual.PositionSize
```

It is a good model to test if the file readers are working correctly, i.e. the application will show a ball of expected size and color that behaves in correctly throughout the animation.

The Pendulum3D is a more complex model. It contains three objects that should be animated, one sphere and two cubes. It also uses offset and rotation for some objects. This model tests many functions of the application. Testing this application allowed to uncover and fix some bugs.

The application has other functionalities besides playing the animation, these functionalities are controlled graphically i.e. through buttons and sliders. The button testing was very straight forward it was easy to see if they worked or not. The sliders were more complicated to test. I had to adjust the behavior as they were jumping in very big ticks, steps. This was solved by normalizing the slider scale. By multiplying the sliders scale by 1000 the sliders steps become so small that they can’t be noticed. This must be remembered because this multiplication factor will affect some data in the application so division by 1000 must be done. Because the testing was only done on these two models it is possible that there are undiscovered errors that didn’t appear with these two models.

4.10. The widget
**Figure 10:** Application example

<table>
<thead>
<tr>
<th>Buttons/ sliders:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLAY / PAUSE</td>
<td>Starts and pause the simulation movie depending on if it’s running or not.</td>
</tr>
<tr>
<td>RESTART</td>
<td>Restarts the movie</td>
</tr>
<tr>
<td>Progress</td>
<td>Shows how far in the simulation you are at the moment</td>
</tr>
<tr>
<td>Start time</td>
<td>Shows start time of simulation. Normal is 0</td>
</tr>
<tr>
<td>End time</td>
<td>Shows how long the simulation is.</td>
</tr>
<tr>
<td>Progress bar</td>
<td>Shows how long in the movie the animation is</td>
</tr>
<tr>
<td>Slider</td>
<td>Can be used to jump forward or backward in the movie. It also show how far into the animation you are</td>
</tr>
<tr>
<td>Speed</td>
<td>Used to run the simulation faster or slower than normal speed. Range is 0.1 * normal speed to 100 * normal speed.</td>
</tr>
<tr>
<td>Zoom</td>
<td>Used to zoom in or out. Range from 0 to 100, i.e. 0 units from centre to 100 units from centre</td>
</tr>
</tbody>
</table>

When the widget is started for a first visualization it opens a QtMainWindow and a visualizeWindow that is a QWidget with some extra functionality. If now a second visualization is started you can as user choose if you want to launch it as a second visualizeWindow or if you want to erase the old and start the new in its place. The visualizationWindow will start up with the simulated object in centre of scene and with a zoom factor that ensures the object fits the screen. The speed of the animation is set to normal i.e. set to run as fast as it is supposed to be in the simulation. The animation starts paused at time 0.

The widget can handle as many objects as the user want. It takes a number of specific shapes, see Appendix 2. Allowed shapes. These shapes are those that the OpenSceneGraph can use by default. The simpleVisual only supports cube, sphere and cylinder shapes so with OMNotebook only these three can be used. OpenSceneGraph supports RGB colors but in the OMVisualize implementation only a fixed number of colors could be used. See Appendix 3. This is chosen to make implementation easier but with some changes in the implementation it should be possible to use the RGB color scheme instead of a supported color. Some changes must also be done in the visualize file reading and the creation of the shapeDrawable so it support it.

When the program is started it is done by typing:

```text
--visualizationfile=arg1 --simulationfile=arg2 --title=arg3 --new-window=arg4
```

The arg1 is the .visualize filename and the arg2 is the animation filename, the .plt, .csv or .mat file. Arg3 is optional and it can be used to set a user specified name to the window. Arg4 is also optional and is a bool that if it’s set to true a new simulation will get its own window but if it’s set to false the new simulation will get the focused windows name and print over it.
5. Result

In this chapter will the problems set for the report be answered.

There exist many animation libraries that support OpenGl. There are also many that like OpenSceneGraph use the technology of scene graph to create animations. Not all are free and open, that’s a big advantage for the OpenSceneGraph that it’s both free and open. During the creation of OMVisualize I found out that it is easy to find information and help about OpenSceneGraph on many forums and internet sites. This is very useful during programming with the OpenSceneGraph library.

5.1. Is the OpenSceneGraph a library that supports good methods?

The OpenSceneGraph supports the programmer by giving many possibilities to program OpenGl by simpler commands than by programming directly with OpenGl commands. OpenSceneGraphs usage of the scene graph technique makes it efficient and modern. The scene graph technique is as I see it going to be used for many years forward and is still being developed.

5.2. Is the OpenSceneGraph useful for the OpenModelica?

The OpenSceneGraph supports animation in a good way. So the package would suit as an animation library for the OpenModelica. The application this thesis resulted in show that it is possible to build an animation tool for OpenModelica with help from the OpenSceneGraph library. The portability is useful for the OpenModelica application, OpenModelica is used by people on different platforms.
6. Analysis

OpenSceneGraph is easy to use in applications because:

- it handles low level graphics automatically
- many users create new and usable functions
- manuals and tips are available online
- it is still being developed so new version appears now and then

The animation path is easy to use if the animation data is organized right. This is an easy way to create a simple animation in a very natural way because I think it’s natural to think in the way that an object is in a specific location at a specific time. The programmer doesn’t have to consider how fast objects move because the OpenSceneGraph calculates speed and direction from the times and places the user have defined. The simple shapes that the library supports are good to use in technical applications, many technical shapes could be build by just these simple shapes so no need to create own shapes.

There are many users that use the OpenSceneGraph so there are many usable implementations to be inspired of or to use in your own code. Forums exist online both the official as well as some nonofficial so help could probably be found there if you can’t find it in the official manual online. The file QOSGGraphics I have used is recycled from earlier development. QOSGGraphics provides functions that connect Qt and OpenSceneGraph in a good way that supports much functionality from both sides.

The OpenSceneGraph and Qt libraries are platform independent so the OMVisualize is also platform independent only depending on the libraries and the compiler used to make it.
7. Future work

If anyone wants to continue the work on the OMVisualize there are some upgrades that could be useful. Support for more colors i.e. use of RGB color scheme. The possibility to use textures on objects could be useful. More color alternatives are maybe a good add on to create.

Interaction with the animation window in more ways could be useful. To rotate up and down as well as right and left could make it easier to watch a simulation. As it is now user could use drag and click by mouse in animation window to do these actions but buttons or sliders could make these actions easier to access.

A better communication between Modelica simulation and OMVisualizer could be useful. In this case faster means better.
8. Summary
In this chapter the accomplishments of the thesis will be summed. The result and the work on the thesis will also be discussed.

8.1. Result
This thesis resulted in the OMVisualize that could be used from the OMNotebook. The OMVisualize could visualize a simulation of models made by OpenModelica that use the simpleVisual library. The OMVisualize could handle the three simulation result file formats that OpenModelica could deliver. It also resulted in the visualize file that is used for communication from OpenModelica to the OMVisualize. The OMVisualize have functions that are useful for animation of simulations. The OMVisualize have functionality to change speed, search to a specific time, play, pause, restart and control the camera in the animation by using the mouse. By making this application it’s showed that it is possible to use the OpenSceneGraph library in the OpenModelica. The OpenSceneGraph library is efficient and good to use.

8.2. Conclusion
This thesis included studies in how to get a small program, the OMVisualize, to work in a large context. The work could be divided into three parts. These parts are the Qt programming, the OpenSceneGraph library and how to get those two to work together with OMNotebook. The first problem I found was to build the OpenSceneGraph library so it worked with my developing environment. This was something I didn’t expect because I never had to do it earlier. Now I know better and found that similar problems are quite common in development. The visualization with use of OpenSceneGraph was the first and easiest part to do. Quite early in the work I had a functional visualize window that could show basic shapes in a variety of colors moving around following specified paths. To use Qt functionality in this visualization window was harder but after some theory studies and testing I found a solution. I wasn’t the first developer that needed the connection between Qt and OpenSceneGraph. OpenSceneGraph forums were studied and I found code that I reused. It was a new and good experience to reuse and find out the special challenges that comes with it. To get the visualization Qt widget to work with OMNotebook it required studies in how to retrieve data and in what format from OpenModelica simulation. To study old code and find what’s wanted is a good experience that I take with me from this work.
Appendix A

Allowed shapes

- Ball and sphere creates a sphere
- Box and cube creates a cube
- Capsule
- Cone
- Cylinder
Appendix B

Allowed colors
The float numbers are red, green, and blue. The last float is always 1.0.

<table>
<thead>
<tr>
<th>Color</th>
<th>osg::Vec4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue</td>
<td>osg::Vec4(0.0f, 0.0f, 1.0f, 1.0f)</td>
</tr>
<tr>
<td>Red</td>
<td>osg::Vec4(1.0f, 0.0f, 0.0f, 1.0f)</td>
</tr>
<tr>
<td>Green</td>
<td>osg::Vec4(0.0f, 1.0f, 0.0f, 1.0f)</td>
</tr>
<tr>
<td>Yellow</td>
<td>osg::Vec4(1.0f, 1.0f, 0.0f, 1.0f)</td>
</tr>
<tr>
<td>Black</td>
<td>osg::Vec4(0.0f, 0.0f, 0.0f, 1.0f) This color looks very bad in animation because the shadows can’t be seen it doesn’t look 3D. It looks flat 2D.</td>
</tr>
<tr>
<td>White</td>
<td>osg::Vec4(1.0f, 1.0f, 1.0f, 1.0f)</td>
</tr>
</tbody>
</table>
Appendix C

How to run OMVisualize

Here it will be explained how to start and use the OMVisualize. I have tested this in OMNotebook 3.0 connected to OpenModelica 1.9.0 Beta 2.

The following example will show how to get your simulation in OMVisualize:

```model BouncingBall3D
  parameter Real e=0.7 "coefficient of restitution";
  parameter Real g=9.81 "gravity acceleration";
  Real h(start=10) "height of ball";
  Real v "velocity of ball"
  Boolean flying(start=true) "true, if ball is flying"
  Boolean impact;
  Real v_new;
  SimpleVisual.PositionSize obj "color=red;shape=sphere;"
  equation
    impact=h <= 0.0;
    der(v)=if flying then -g else 0;
    der(h)=v;
    obj.size[1]=5;
    obj.size[2]=5;
    obj.frame_a[1]=0;
    obj.frame_a[2]=h+obj.size[2]/2;
    obj.frame_a[3]=0;
    when {h <= 0.0 and v <= 0.0,impact} then
      v_new=if edge(impact) then -e*pre(v) else 0;
      flying=v_new > 0;
      reinit(v, v_new);
  end when;
end BouncingBall3D;

loadModel(SimpleVisual)
simulate(BouncingBall3D, startTime=0, stopTime=10)
visualize(BouncingBall3D)
```

The OMVisualize could also be run with the Qt. Open the OMVisualizeGUI.pro file. To run the OMVisualize in Qt use the command line:

```
--visualizationfile=fileName.visualize --simulationfile=fileName.mat --
title=someName --new-window=true
```

The filename.visualize is the link to the visualize file. The filename.mat is the link to the simulation file, it could also have file extension plt or csv. The title is used to set a name on the animation. If new-window is set to true all new animations will be added as new window in OMVisualize. If it is set to false new animations will close earlier animation before opening, it will result in only one animation being showed.
Bibliography


18. OpenSceneGraph graphics engine, (accessed 2012-08-25), [www]<http://www.openscenegraph.org/documentation/OpenSceneGraphReferenceDocs/a00014.html#3bcd177d9e8732910353b949b9a2af0b>