3D Animation and Programmable 2D Graphics for Visualization of Simulations in OpenModelica

Henrik Eriksson, Henrik Magnusson, Peter Fritzson, Adrian Pop

PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, S-581 83 Linköping, Sweden
petfr@ida.liu.se

Abstract

This paper describes recent work on visualization of simulation results from simulating Modelica models in Open-Modelica. A new 3D graphics package with interactive animation and a new flexible programmable 2D graphics have been added to OpenModelica. The 2D graphics package provides very flexible usage, either directly from a simulation, from the electronic book client OMNotebook, or programmable graphics, called directly from a Modelica model.

1 Need for More Flexible 2D Graphics

1.1 Limitations of Previous Plotting Approach

OpenModelica [1][4] has previously used the external program PtPlot [6] for generation of 2D graphics from simulation data generated from Modelica [2][3] model simulations. Although this program is well suited for this kind of tasks, this solution has some flaws. Firstly, PtPlot is written in Java, which means that the Java Virtual Machine has to be installed for OpenModelica to work as intended. Furthermore, integration with OMNotebook [10][11] is difficult which has led to the current situation where diagrams are treated as images. This makes inter-activity, e.g. zooming, scaling or suchlike, impossible for diagrams that have been inserted into a document. PtPlot is only available from the API of OMC (OpenModelica Compiler) and cannot be used from within Mode-lica models. This prevents for example models that create diagrams or other kinds of graphics.

The 2D plotting deals mainly with the most common types of diagrams, two-dimensional line diagrams and two-dimensional parametric diagrams, and variations of these. A more complete graphics package should for ex-ample contain more types of diagrams and provide means to create animations, but implementation of this kind of features is future work..

1.2 New 2D Graphics Package

In order to solve the problems mentioned above a new graphics package [8] has been developed. Parts connected to OMNotebook has been developed in the same environment as this, i.e. in C++ and Qt. CORBA is still used to send commands from OMNotebook to OMC, but instead of calling PtPlot to create diagrams OMC sends the simulation data to OMNotebook, which handles the presentation (see Figure 1). As OMNotebook now has access to all source data it will now be possible to manipulate diagrams, e.g. zoom or change scales.

To allow the use of graphics functions from within Modelica models a new Modelica interface has been devel-oped. This utilizes an external library to communicate with OMNotebook. In addition to this, a number of new functions that can be used for drawing geometric objects like circles, rectangles and lines have been added.

The main goal of this work has been to extend the data visualization capabilities of OpenModelica. This is ac-complished by solving the problems of the previous solution, but also by introducing new ways to draw diagrams and other types of graphics from within Modelica models. The following is a summary of the capabilities of the new 2D graphics package:

- *Interaction with OMNotebook*. The graphics package has been developed to be fully integrated with OM-Notebook and allow modifications of diagrams that have been previously created.
- *Usage without OMNotebook*. If the functionality of the graphics package is used without OMNotebook, a new window should be opened to present the resulting graphics.
- *Logarithmic scaling*. Some applications of OpenModelica produce simulation data with large value ranges, which is hard to make good plots of. One solution to this problem is to scale the diagram logarithmically, and this is allowed by the graphics package.
- *Zoom*. To allow studying of small variations the user is allowed to zoom in and out in a diagram.
- *Support for graphic programming*. To allow creation of Modelica models that are able to draw illustrations, show diagrams and suchlike, it is possible to use the graphics package not only from the external API of OMC, but also from within Modelica models. To accomplish this a new Modelica interface for the graphics package has been created.
- *Programmable Modelica API*. The Modelica API is defined by a number of Modelica functions, located in the package *Modelica.Utils.Graph*, which use external libraries to access functionality of the graphics package. These functions include the following:

  - `plot`(x). Draws a two-dimensional line diagram of *x* as a function of time.
  - plotParametric(x,y) Draws a two-dimensional parametric diagram with *y* as a function of *x*.
  - `plotTable`([x_1, .., y_1; .. ; x_n, .., y_n]). Draws a two-dimensional parametric diagram with *y* as a function of *x*.
  - `drawRect`(x_1, x_2, y_1, y_2). Draws a rectangle with vertices in (x_1, y_1) and (x_2, y_2).
  - `drawEllipse`(x_1, x_2, y_1, y_2). Draws an ellipse with the size of a rectangle with vertices in (x_1, y_1) and (x_2, y_2).
  - `drawLine`(x1, x2, y1, y2). Draws a line from (x_1, y_1) to (x_2, y_2).

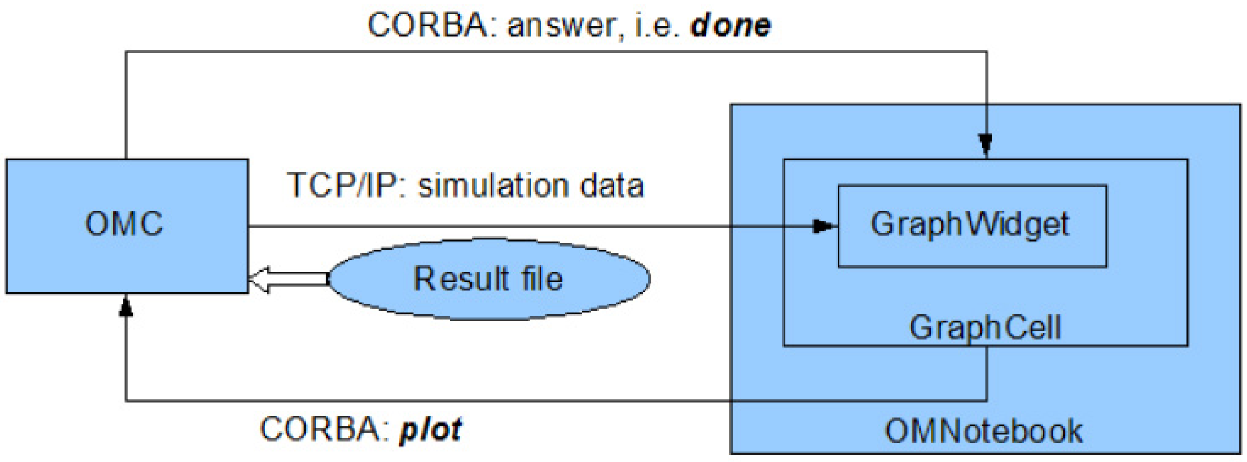


Figure 1. Plotting architecture with the new 2D graphics package.

1.3 TCP/IP Based Data Transfer

A new TCP/IP-based interface has been developed and is used to transfer simulation data from OMC to OMNotebook. This is built with socket classes from Qt and also allows simulation data to be transferred during a running simulation (see Figure 2). Usage of Qt classes has some drawbacks, OMC depends on Qt libraries for example, but there are also advantages. Qt provides, for instance, powerful ways to serialize data and sockets that can be used to transfer these data streams. Besides, Qt is already required to build OMNotebook and OMShell, and the necessary runtime libraries are already distributed with the OpenModelica setup program (for Windows). Drawing of diagrams and other kinds of graphics is done using the classes *QGraphicsScene* and *QGraphicsView*, which were introduced in Qt 4.2. These provides methods to create and present various kinds of graphic primitives, e.g. lines, circles and suchlike, but also more complex objects can be generated.

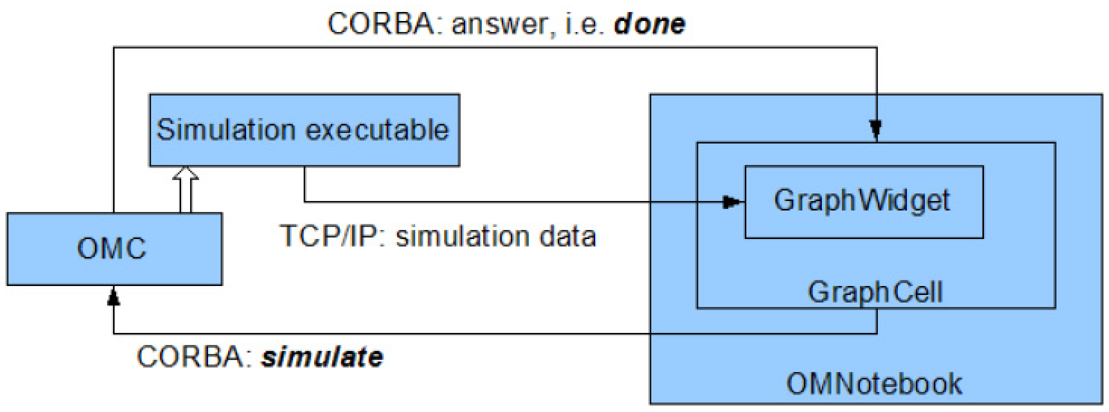


Figure 2. Data transfer during simulation using the new plotting package.

Two types of data transfer via TCP/IP have been implemented, one for sending generated simulation data during a running simulation and one for sending required data when specific commands, such as *plot* or *plotParametric*, are given. These types have many things in common, but both are required as it probably is desirable for instance to be able to plot data without having to run a previously run simulation again. The latter type also includes parameters that govern how the generated diagram will appear.

TCP/IP communication is also used to implement the Modelica interface of the graphics package. This consists of functions like *drawEllipse*, which when evaluated will use a socket to send its parameters to a receiving GraphWidget.

The different modes of data transfer are identified by a text string that is sent in the first package of a data stream. The following modes have been implemented:

- `simulationDataStream`. Indicates that the following data stream contains simulation data that should be stored in the active GraphWidget for later use.
- `ptolemyDataStream`. Indicates that the following data stream contains simulation data and formatting information required to immediately create a diagram.
- `drawEllipse`. Indicates that the package contains information necessary to draw an ellipse in the active GraphWidget.
- `drawLine`. Like drawEllipse, but for a line.
- `drawPoint`. Like drawEllipse, but for a point.
- `drawRect`. Line drawEllipse, but for a rectangle.
- `drawText`. Indicates that the package contains information necessary to print text in the active GraphWidget.
- `clear`. Indicates that the active GraphWidget should be reset.
- `closeServer`. Indicates that the active GraphWidget should stop listening on the current port. This is used to control which GraphWidget that is considered active.
- `hold`. Indicates that the active GraphWidget should not be cleared before new graphics is shown.

The following commands have been added to the API of OMC in order to control the data transfer interface:

- `enableSendData`(Boolean). Determines whether or not simulation data should be transferred during simulation.
- `setDataPort`(int). Determines which port the data transfer interface should use.
- `setVariableFilter`("x_1", "x_2", . . . , "x_n"). Determines the variables that should have their values sent during simulation. An empty string indicates that all variables will be sent.

2 The GraphWidget

GraphWidget is the name of the new widget that has been developed to create and present graphics. GraphWidget is derived from the Qt class *QGraphicsView* and thereby inherits an area where graphics can be drawn. In addition to this, functionality has been added for instance to handle TCP/IP communication with OMC.

2.1 Reception of Simulation Data

Reception of simulation data will occur if the word *simulationDataStream* is the first in the identifying package sent by OMC. The values that are received are stored internally in the active GraphWidget and can later be used for data plotting by clicking on *Simulation data* in the context menu of the GraphWidget or by clicking the button *D*, which appears to the right of the input field when the simulation data has been transferred.

2.2 Drawing of Diagrams

Drawing of diagrams can be initiated in two ways. Either by sending commands like *plot* or *plotParametric* to OMC, which responds by sending a data stream described in Sections2.2.1 1.3 and , or by using the dialog *Simulation data* to create diagrams from simulation data stored in the GraphWidget.

2.2.1 Drawing from a Data Stream

If the identifying keyword of the data stream from OMC is *ptolemyDataStream*, a diagram will be created instantly from the stream, which besides simulation data contains variable names, diagram titles and other information necessary to generate a plot. For each pair of *x* and *y* variables, typically a time value and a variable value of that time, a *QGraphicsEllipseItem* object is created and inserted into the *QGraphicsScene* object of the GraphWidget. The visibility of this object is controlled by the *drawPoints* parameter of the plot command. If a previous data point has been received, *QGraphicsLine* objects are then created to connect the two most recent data points according to the interpolation mode that is selected. The visibility of these lines is determined by the *drawLine* parameter. The received simulation data is also stored internally in the GraphWidget and can later be used for instance to create new curves or to scale the diagram logarithmically.

2.2.2 Drawing from Stored Data

This procedure is essentially identical to that described in Section 2.2.1. The data that is used is however not received through a data stream but instead taken from internal tables, where it has previously been stored.

2.3 Drawing of Geometrical Objects

The commands provided by the Modelica interface for drawing of geometrical objects require, unlike the *plot* and *plotParametric* commands, no simulation data to be sent. All necessary information is given as parameters (default values are used if no values are specified). When OMC receives a command like this, a data stream is created, and the parameters are sent through this in a packet like the following.

```
(QString) id
(double) x0
(double) y0
(double) x1
(double) y1
(QColor) color
(QColor) fill color
```

The string *id* specifies the purpose of the transmission. See Section 1.3 for a list of possible values.

2.4 External Usage

As the GraphWidget is a widget, it is easy to use in applications other than OMNotebook. This has been used to enable presentation of diagrams when functions of the graphics package are used from other applications than OMNotebook, for instance by running a script. If a command like *plot* or *plotParametric* is given, or if a model using functions of the graphics package is simulated it is desirable that a window showing the diagram is displayed even if OMNotebook is not running. This problem has been solved by developing new application, *ext.exe*, which is essentially just a GraphWidget in a dialog window. This application is launched if OMC fails to connect to a GraphWidget when a command of the graphics package has been received.

3 2D Plot Examples

3.1 Simple 2D Plot

To create a simple time plot the model `HelloWorld` defined in DrModelica is simulated. To reduce the amount of simulation data in this example the number of intervals is limited with the argument `numberOfIntervals=10`. The simulation is started with the command below.

```
simulate(HelloWorld, startTime=0, stopTime=4, numberOfIntervals=10);
```

When the simulation is finished the file `HelloWorld res.plt` contains the simulation data. The contents of the file is the following (some formatting has been applied).

```
0                      1
4.440892098500626e-013 0.9999999999995559
0.4444444444444444     0.6411803884299349
0.8888888888888888     0.411112290507163
1.333333333333333      0.2635971381157249
1.777777777777778      0.1690133154060587
2.222222222222222      0.1083680232218813
2.666666666666667      0.06948345122279623
3.111111111111112      0.04455142624447787
3.555555555555556      0.02856550078454138
4                      0.01831563888872685
```

Diagrams are now created with the new graphics package by using the following command.

```
plot(x);
```

The diagram shown in Figure 3 seems to correspond well with the data.

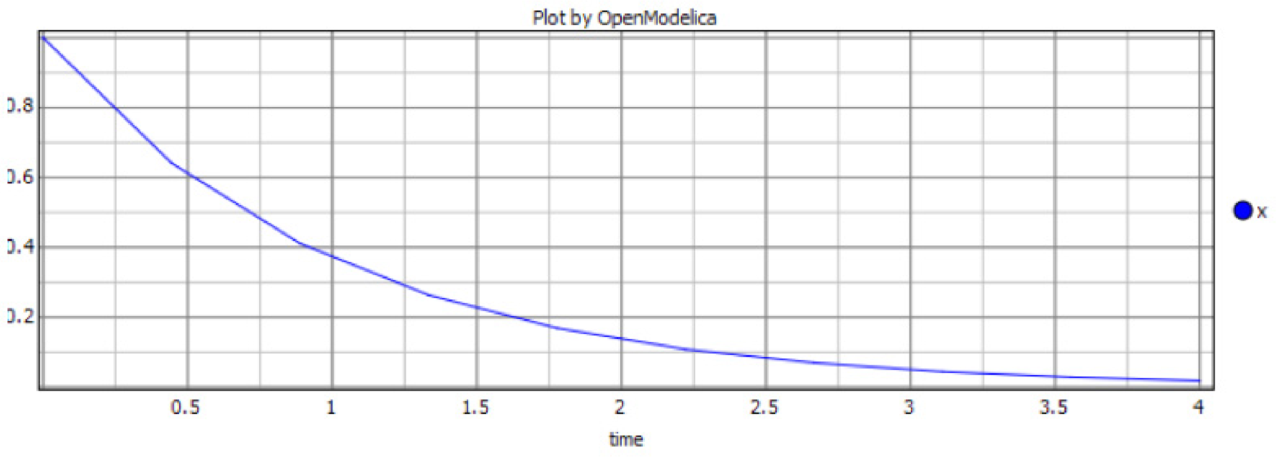


Figure 3. Simple 2D plot of the HelloWorld example.

3.2 Plotting During Simulation

When running long simulations, or if plotting without need for commands like *plot* or *plotParametric* is desired, the interface for transfer of simulation data during running simulations can be used. This is enabled by running the following command.

```
enableSendData(true)
```

The same command, but with the parameter *false*, is used to disable the interface. Enabling of the interface has some drawbacks though. The simulation time will be longer as the transfer of data will require some resources.

If the simulation data would have been plotted anyway, some of this time will be saved later however. To reduce the amount of data that has to be transferred, and thereby reduce the time needed to do so, the interesting variables in the model can be specified with the command *setVariableFilter*. If for instance the model *HelloWorld* is to be simulated the following commands can be used.

```
class HelloWorld
  Real x(start = 1);
  parameter Real a = 1;
equation
  der(x) = - a * x;
end HelloWorld;

enableSendData(true);
setVariableFilter({x});
simulate(HelloWorld, startTime=0, stopTime=25);
```

When the simulation data has been transferred the button *D* will appear to the right of the input field. By pressing this the dialog *Simulation data*, where new curves can be created.

3.3 Programmable Drawing of 2D Graphics

The graphics package provides functions for drawing of basic geometrical objects in the graphics area. These can be used from Modelica models and are executed when the model is simulated. The following model shows how these functions can be used to draw ellipses, rectangles, and lines.

```
model testGeom
  parameter Integer n=10;
  protected
  Boolean b[n,n];
equation
  for x in 1:n loop
    for y in 1:n loop
      when initial() then
        if((y == 1) or (y == 10) or (x == 1) or (x == 10)) then
          b[x,y] = pltpkg.rect(x, y, x+1, y+1, fillColor = "blue",
          color = "green");
        else if(y >= 4 and y <= 5 and x >= 4 and x <= 5) then
            b[x,y] = pltpkg.line(x, y, x+1, y+1, color = "red");
          else
            b[x,y] = pltpkg.ellipse(x, y, x+1, y+1, fillColor = "yellow",
            color = "black");
          end if;
        end if;
      end when;
    end for;
  end for;
end testGeom;
```

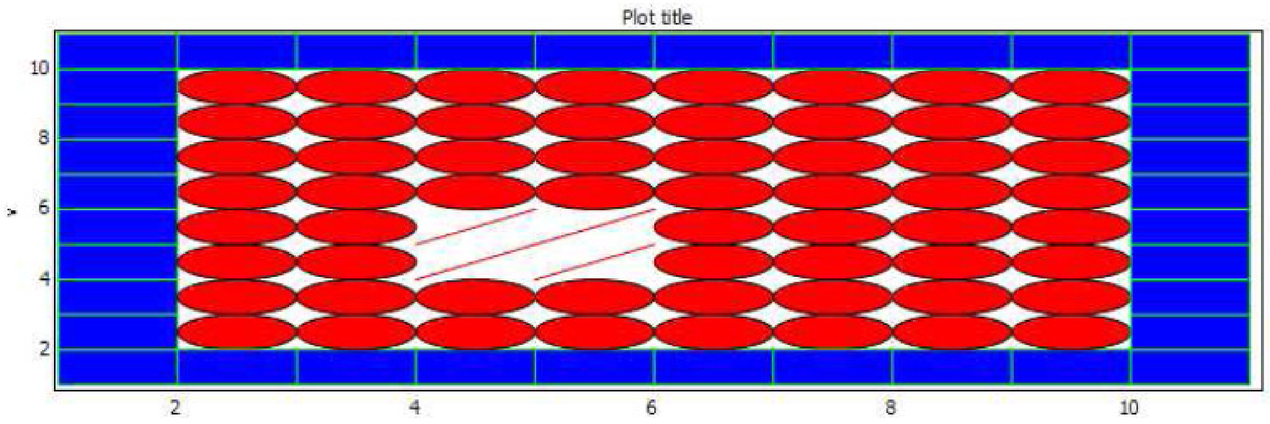


Figure 4. Programmable drawing of rectangles and ellipses.

4 Associating 3D Graphics Information with Modelica

. There are two main approaches to add 3D graphics information to Modelica objects:

- Graphical annotations
- Graphical objects

Both of these approaches were investigated, but the second was finally chosen.

4.1 Visualization Annotation

None of the existing Modelica annotation types are suited for describing visualization information. To overcome this problem, a new visualization type has to be defined. What is needed is to connect a property of the visual representation with a variable in the simulation. For example, if we are visualizing a falling object, the variable containing the height of the object must be connected to the y position of a visualization.

```
annotation(Visualization(frame_a="x", frame_b="y", color="red", shape="cube"))
```

A trial implementation of the annotation approach was done, but some serious flaws was discovered with the approach and the implementation was discarded.

- The annotation system is very inflexible. Querying the OpenModelica compiler for an annotation yields only a list of values with no specification of the contents. To support many different kinds of visualizations, i.e. displaying cubes, spheres, with or without color etc, a new annotation would have to be defined for each type. Since there is no polymorphism in the annotation syntax, this quickly becomes problematic.
- Type checking is impossible when the only connection between a property (such as the position of an object) and the variable containing the information is the name of the variable. In the example above, there is no guarantee that x is of a type that is usable as a position.
- Syntax checking is not performed on annotations in the sense that it is impossible to know at compile time. if the visualization annotation is correctly specified.

4.2 Object Based Visualization

Since one important goal of this work is to come up with a system for visualization that might be used for simulations done with the Modelica MultiBody library [12], it follows that much can be learned from investigating currently available solutions. There are commercial software packages available that can visualize MultiBody simulations.

The MultiBody package is well suited for visualization. Entities in a MultiBody simulation correspond to physical entities in a real world and as such have many of the properties needed to correctly display them within a visualization of the simulation, such as position and rotation. Other properties such as color and shape can easily be added as properties or be decided based on the object type.

Instead of using annotations to encode information about how a certain object is supposed to look when visualized, object based visualization creates additional Modelica objects of a predetermined type that can be known to the client actually doing the visualization. These objects contain variables such as position, rotation and size that can be connected to the simulated variables using ordinary Modelica equations. When asked to visualize a model, the OpenModelica compiler can find variables in the model that are in the visualization package and only send only those datasets over to the client doing the visualization, in this case OMNotebook.

Taking inspiration from the MultiBody library, a small package has been designed that provides a minimal set of classes that can be connected to variables in the simulation. It is created as a Modelica package and can be included in the Modelica Library. The package is called `SimpleVisual`, and consists of a small hierarchy of classes that in increasing detail can describe properties of a visualized object. It is implemented on top of the Qt graphics package called Coin3D [7]. More information is available in [9]. A comprehensive earlier work on integrating and generating 3D graphics from Modelica models is reported in [13].

5 3D Graphics Examples

This section gives a short introduction to how the SimpleVisual package is used.

5.1 BouncingBall

The bouncing ball model is a simple example to the Modelica language. Adding visualization of the bouncing ball using the SimpleVisual package is very straightforward.

```
model BouncingBall
  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;
equation
  impact=h <= 0.0;
  der(v)=if flying then -g else 0;
  der(h)=v;
  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 BouncingBall;
```

To run a simulation of the bouncing ball, create a new InputCell and call the simulate command. The simulate command takes a model, start time, and an end time as arguments.

```
simulate(BouncingBall, start=0, end=5s);
```

5.1.1 Adding Visualization

The bouncing ball will be simulated with a red sphere. We will let the variable h control the y position of the sphere. Since the ball has a size and the model describes the bouncing movement of a point, we will use that size to translate the visualization slightly upwards. First, we must import the SimpleVisual package and create an object to visualize. That is done by adding a few lines to the beginning of the BouncingBall model.

```
model BouncingBall
import SimpleVisual.*
SimpleVisual.PositionSize ball "color=red;shape=sphere;";
...
```

The string "color=red;" is used to set the color parameter of the object and the shape parameter controls how we will display this object in the visualization.

The next step is to connect the position of the ball object to the simulation. Since Modelica is an equation based language, we must have the same number of variables as equations in the model. This means that even though the only aspect of the ball that is really interesting is its y-position, each variable in the ball object must be assigned to an equation. Setting a variable to be constant zero is a valid equation. The SimpleVisual library contains a number of generic objects which gives the user an increasing amount of control.

```
SimpleVisual.Position
SimpleVisual.PositionSize
SimpleVisual.PositionRotation
SimpleVisual.PositionRotationSize
SimpleVisual.PositionRotationSizeO_set
```

Since we are really only interested in the position of the ball, we could use `SimpleVisual`.Position, but to make it a little bit more interesting we use `SimpleVisual.PositionSize` and make the ball a little bigger.

```
obj.size[1]=5;  obj.size[2]=5;  obj.size[3]=5;
obj.frame_a[1]=0;  obj.frame_a[2]=h+obj.size[2]/2;  obj.frame_a[3]=0;
```

A `SimpleVisual.PositionSize` object has two properties; `size` and `frame_a`. All are three dimensional real numbers, or Real[3] in Modelica.

- `size` controls the size of the visual representation of the object.
- `frame_a` contains the position of the object.

5.1.2 Running the Simulation and Starting Visualization

To be able to simulate the model with the added visualization, OpenModelica must load the SimpleVisual package.

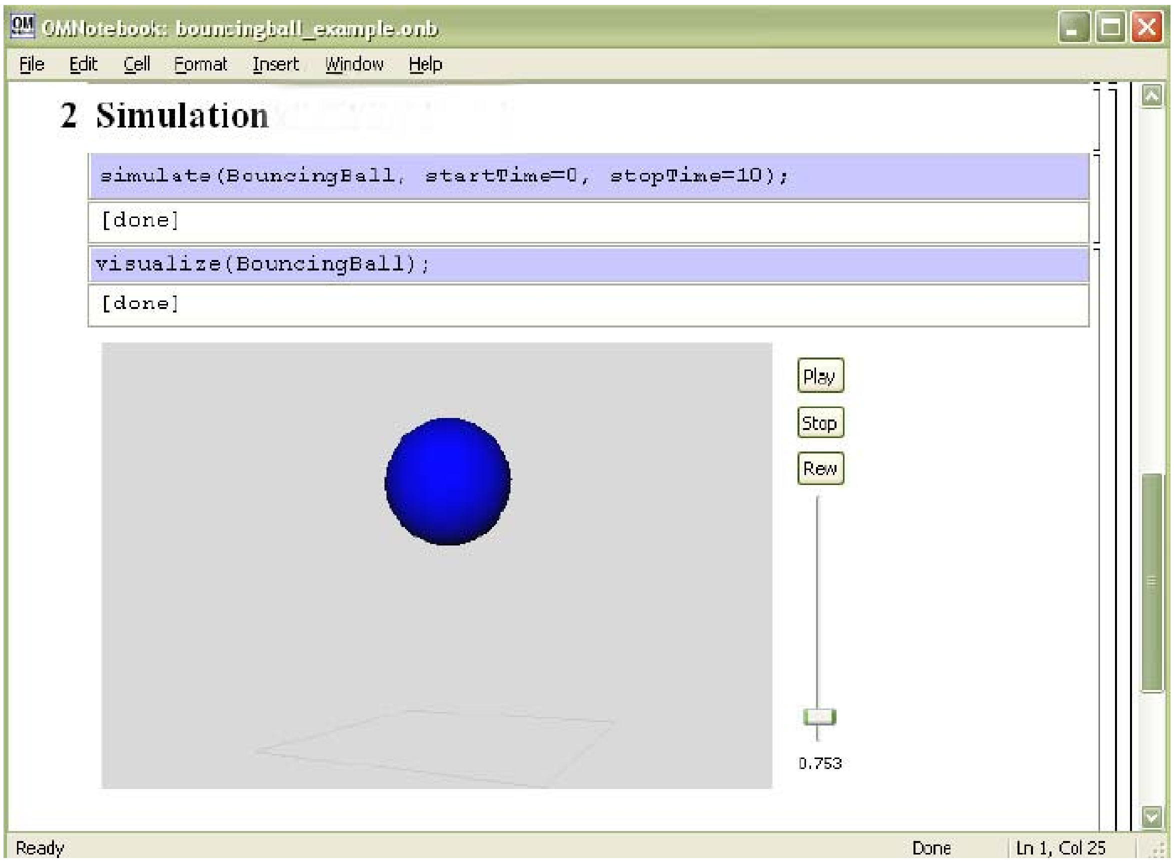
```
loadLibrary(Modelica.SimpleVisual)
```

Now, call simulate once more. This time the simulation will generate values for the added SimpleVisual object that can be read by the visualization in OMNotebook.

```
simulate(BouncingBall, start=0, end=5s);
```

To display the visualization, create an input cell and call the visualize in the input part of the cell.

```
visualize(BouncingBall);
```



```
simulate(BouncingBall, startTime=0, stopTime=10);
[done]
visualize(BouncingBall);
[done]
```

Figure 5. 3D animation of the bouncing ball model.

5.2 Pendulum 3D Example

This example explores a slightly more complex scenario where the visualization uses all the properties of a SimpleVisual object. The model used is a simple ideal 2D pendulum, not modeling properties like friction, air resistance etc.

```
class MyPendulum "Planar Pendulum"
constant Real PI=3.141592653589793;
parameter Real m=1, g=9.81, L=5;
Real F;
Real x(start=5),y(start=0);
Real vx,vy;
equation
  m*der(vx)=-(x/L)*F;
  m*der(vy)=-(y/L)*F-m*g;
  der(x)=vx;
  der(y)=vy;
  x^2+y^2=L^2;
end MyPendulum;
```

Start by identifying the variables in the model that will be needed to create a visual representation of the simulation.

- Real x and Real y hold the current position of the pendulum.
- Real L is a parameter which holds the length of the pendulum.

5.2.1 Adding the Visualization

As before, to be able to use the SimpleVisual package we must import it.

```
class MyPendulum "Planar Pendulum"
import Modelica.SimpleVisual;
...
```

Adding a sphere to represent the weight of the pendulum is done in the same way the BouncingBall was visualized. The variables x and y hold the position.

```
...
Real vx,vy;
SimpleVisual.PositionSize ball "color=red;shape=ball;";
equation
  ball.size[1]=1.5;  ball.size[2]=1.5;  ball.size[3]=1.5;
  ball.frame_a[1]=x;  ball.frame_a[2]=y;  ball.frame_a[3]=0;
  m*der(vx)=-(x/L)*F;
...
```

The next step is to create a visualization of the "thread" that holds the pendulum. It will be represented by a small elongated cube connected to the ball in one end and in the fixed center of the pendulum movement. We will want the object to rotate with the pendulum motion so create a `SimpleVisual.PositionRotationSize` object.

```
SimpleVisual.PositionRotationSize thread "shape=cube;"
```

To specify the rotation of an object, the visualization package uses two points. One is the position of the object, frame_a, that has been demonstrated earlier. The other position, `frame_b`, is interpreted as the end point of a vector from frame_a. This vector is used as the new up direction for the object. In this example, defining `frame_b` is simple. The cube that represents the thread will always be pointing to (0, 0, 0). We already know the length of the thread from the parameter L.

```
thread.size[1]=0.05;  thread.size[2]=L;  thread.size[3]=0.05;
thread.frame_a[1]=x;  thread.frame_a[2]=y;  thread.frame_a[3]=0;
thread.frame_b[1]=0;  thread.frame_b[2]=0;  thread.frame_b[3]=0;
```

Running this simulation and starting the visualization, we notice that everything is not quite right. The thread is centered around the pendulum. We could calculate a new position by translating the x and y coordinates along the rotation vector, but there is a better way. Change the object type to `SimpleVisual.PositionRotationSizeOffset`. The offset parameter is a translation within the local coordinate system of the object. To shift the center of the object to be at the bottom of the thread we add an offset of L/2 to the y component of offset.

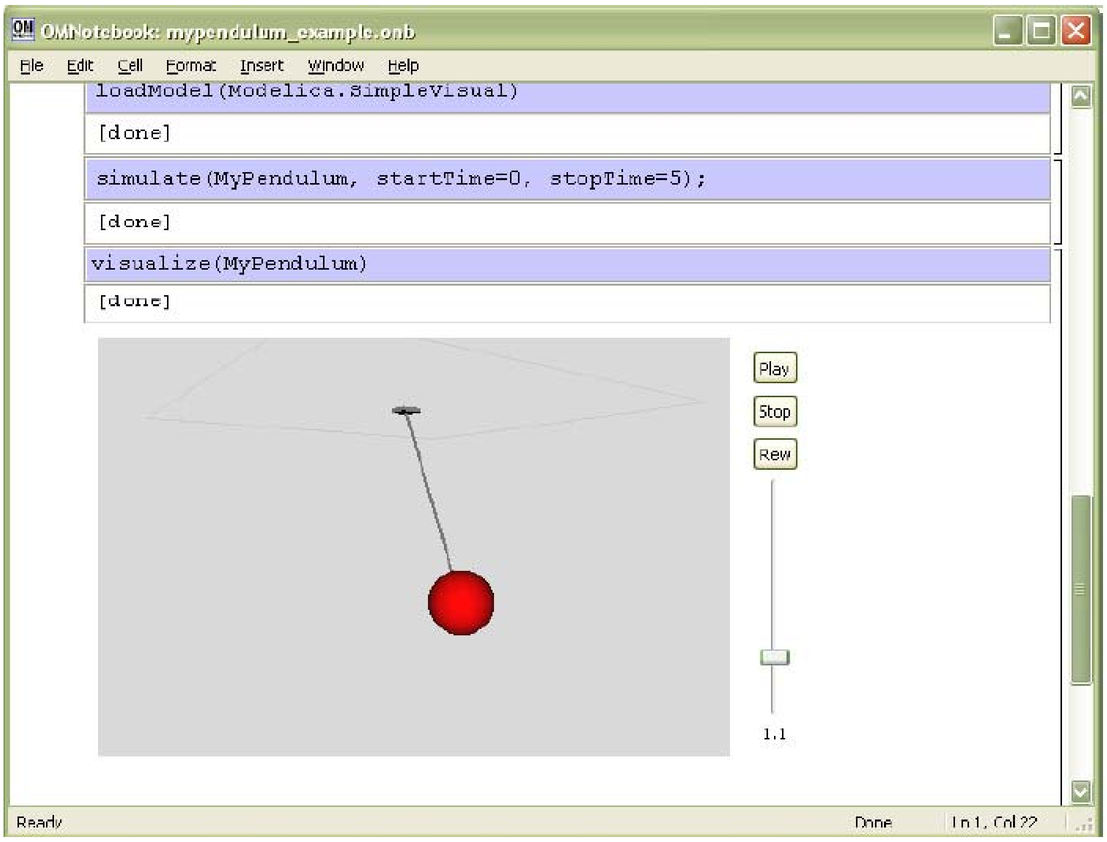
```
thread.size[1]=0.05;  thread.size[2]=L;  thread.size[3]=0.05;
thread.frame_a[1]=x;  thread.frame_a[2]=y;  thread.frame_a[3]=0;
thread.frame_b[1]=0;  thread.frame_b[2]=0;  thread.frame_b[3]=0;
thread.offset[1]=0;  thread.offset[2]=L/2;  thread.offset[3]=0;
```

In the final model, a simple static fixture has also been added.

```
class MyPendulum "Planar Pendulum"
  import Modelica.SimpleVisual;
  constant Real PI=3.141592653589793;
  parameter Real m=1, g=9.81, L=5;
  Real F;
  Real x(start=5),y(start=0);
  Real vx,vy;
  SimpleVisual.PositionSize ball "color=red;shape=ball;";
  SimpleVisual.PositionSize fixture "shape=cube;";
  SimpleVisual.PositionRotationSizeOffset thread "shape=cube;";
equation
```

```
    fixture.size[1]=0.5; fixture.size[2]=0.1; fixture.size[3]=0.5;
    fixture.frame_a[1]=0; fixture.frame_a[2]=0; fixture.frame_a[3]=0;
    ball.size[1]=1.5; ball.size[2]=1.5; ball.size[3]=1.5;
    ball.frame_a[1]=x;  ball.frame_a[2]=y;  ball.frame_a[3]=0;
    thread.size[1]=0.05;  thread.size[2]=L; thread.size[3]=0.05;
    thread.frame_a[1]=x; thread.frame_a[2]=y; thread.frame_a[3]=0;
    thread.frame_b[1]=0; thread.frame_b[2]=0; thread.frame_b[3]=0;
    thread.offset[1]=0; thread.offset[2]=L/2; thread.offset[3]=0;
    m*der(vx)=-(x/L)*F;
    m*der(vy)=-(y/L)*F-m*g;
    der(x)=vx;
    der(y)=vy;
    x^2+y^2=L^2;
  end MyPendulum;
```

We simulate and visualize as previously:


OMNotebook: mypendulum_example.onb
File Edit Cell Format Insert Window Help
Play
Stop
Rew
1.1
Ready
Done
Ln 1, Col 22

```
loadModel(Modelica.SimpleVisual)
[done]
simulate(MyPendulum, startTime=0, stopTime=5);
[done]
visualize(MyPendulum)
[done]
```

Figure 6. Visualization with animation of 3D pendulum.

6 Conclusions and Future work

A new 2D plotting package and a new 3D graphics animation package have been designed and implemented for visualizing results from Modelica model simulations within OpenModelica. The 2D plotting package has a new flexible architecture that allows usage either directly from a running simulation, from the OMNotebook, or programmable graphics from within a Modelica model.

7 Acknowledgements

This work was supported by Vinnova in the Safe Modeling and Simulation Project and by SSF in the OpenModelica Community Project.

References

[1] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Software Development Environment. In Simulation News Europe, 44/45, December 2005. See also: http://www.openmodelica.org.

[2] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, 940 pp., ISBN 0-471-471631, Wiley-IEEE Press, 2004.

[3] The Modelica Association. The Modelica Language Specification Version 3.0, Sept 2007. http://www.modelica.org.

[4] Peter Fritzson et al. The OpenModelica Users Guide. www.openmodelica.org.

[5] Trolltech. Qt. http://www.trolltech.com/, accessed July 2007.

[6] The Ptolemy Project. Ptplot. http://ptolemy.berkeley.edu/body.htm, accessed July 2007.

[7] Coin3D. www.coin3d.org, accessed August 2008.

[8] Henrik Eriksson. Advanced OpenModelica Plotting Package for Modelica. Master Thesis, LIU-IDA/LITH-EX-A-08/036-SE, Linköping University Electronic Press, www.ep.liu.se, June 22, 2008.

[9] Henrik Magnusson. Integrated Generic 3D visualization of Modelica Models. Master Thesis, LIU-IDA/LITH-EX-A-08/035-SE, Linköping University Electronic Press, www.ep.liu.se, June 27, 2008.

[10] Anders Fernström. Extending OMNotebook - An Interactive Notebook for Structured Modelica Documents. Master's thesis, Linköping University Electronic Press, www.ep.liu.se, 2006.

[11] Ingemar Axelsson. OpenModelica, Notebook for Interactive Structured Modelica Documents. Master's thesis, Linköping University Electronic Press, www.ep.liu.se, 2005.

[12] Martin Otter. The Modelica MultiBody Library. http://www.modelica.org/libraries/Modelica, Modelica.Mechanics.MultiBody, accessed August 2008.

[13] Vadim Engelson. *Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing*. Ph.D. Thesis. Linköping Studies in Science and Technology, Dissertation No. 627, http://www.ida.liu.se/~vaden/thesis/, 2000.