Contents

1 Background

Modelica[8] is an object-oriented language for modeling dynamic systems. When Modelica models are simulated, variables constrained by algebraic and differential equations evolve in time. The current tools for visualization of Modelica simulation results [4, 7, 6] have facilities to render *static* objects only. These objects are rigid, i.e. they cannot change their shape. Modelica is a perfect tool for modeling physical processes, especially mechanical systems. However, the bodies in these processes are modeled using very simplified geometries. One of the reasons is that there are limited facilities for specification of body geometry and 3D graphical information in Modelica. Either a fixed set of primitive shapes should be used, or external files with shape and graphics description are referenced.

Modelica is a convenient tool for modeling mechanical systems. The Multibody Simulation library (MBS)[10] is intended for this purpose. The dynamic properties of bodies are obviously in dependent on their geometry. This circumstance is only used in MBS for a limited set of bodies with primitive shapes (box, cylinder, etc.). Furthermore, the shape of the bodies is not utilized for collision detection and contact force computation.

Our solution to the problem of integrating Modelica and graphics is to *develop a standard notation* for 3D geometry data in Modelica, and to *incorporate* this information into appropriate classes used in Modelica models. This information will be created during model design, updated when the model changes, used for computation of certain physical quantities (which can vary) of the bodies during simulation, and used for rendering of the bodies during online or offline visualization. Initially this information can either be created manually, or imported from a modeling tool, e.g. a CAD system.

This solution is motivated by several factors:

- Modelica models and their visualizations have the same structure. Therefore this structural consistency should be supported. The geometry of a whole Modelica model is the same as the union of the shapes of all class instances in the model.

- Values can be reused between a Modelica class of a physical object, corresponding geometric construction and 3D graphical model. In particular, constants defined in a geometric model are reused as constants in the Modelica physical model since they are related to the same physical object. Parameters from the Modelica model affect the visualization of the corresponding 3D graphic model.

- Graphics and model code can be kept together by including all the necessary information into the same document. Such documents, together with textual documentation which is already included as a textual annotation in Modelica models, can be used for information exchange between developers.

There are also some disadvantages of inserting geometry descriptions into Modelica models. If a complex geometry is used the model becomes very large, having the following consequences:

- It can be cumbersome to edit the textual representation of the model. However the geometry descriptions can be hidden when connection diagrams are edited, using special model editing environments (MathModelica[7]) or their customization (e.g. special Emacs[5] mode).

- Compilation of models with complex geometries can take too much time. A solution to this problem can be placing geometry descriptions in Modelica code such way that these can be ignored during normal compilation. This issue is discussed in Section 3.

It is important to note that annotations in general, and geometric annotations in particular, *do not change the semantics of Modelica models*. They do not create any new equations or new variables. However, they can affect the execution of Modelica code (if the user specifies function calls that request data based on geometry), in a similar way as input data can affect execution.

2 Simulation and Visualization Requirements

There are several requirements which are essential for the simulation of physical models with non-trivial geometry:

- The volume, mass (when density is given), position of the center of mass (assuming that bodies are homogeneous), and the matrix of inertia should be automatically evaluated for each body.

- In the context of simulations where contacts between bodies are important, collisions can be detected. Collisions between bodies can be detected only if the complete geometric information is given for all the bodies.

- Contact force evaluation: if a collision of bodies is detected, an external force caused by the collision should affect the physical model behavior.

- Partial differential equations (PDEs) are not part of Modelica yet. However it is clear that geometry information related to domain, distribution and grid generation is necessary for solving PDEs. This geometry data should be described in Modelica model somewhere.

For some application areas the geometry associated with physical models can be used for simulation; in some others there is no relation between simulation and geometry at all. In particular, geometry is ignored in electrical modeling. The body geometry is partially used in dynamic analysis of mechanical systems (the kinematic outline is usually sufficient). In chemical processes a few simple shapes, e.g. cylinders are sufficient. Complete and exact geometry descriptions are, however, necessary for computing contact forces between bodies, i.e. friction of surfaces and collision. Complete geometry information is necessary for visualization of physical processes.

The most important features that should be available in the visualization environment are:

- primitive graphic objects with vertex coordinates (point, line, triangle, quadrilateral[1] and polygon) in 2D and 3D; These coordinates can be constants or variables computed during simulation.

- colors;

- objects with fixed shape that change position and orientation over time (rigid body visualization).

Other features that should be available in the environment are:

- normal vectors to surfaces (necessary for correct rendering);

- material illumination and transparency properties;

- texture raster and texture mapping;

- coordinate transformations, such as translation, rotation, scaling (including transformations and rotation matrices such as those available in the MBS library);

- predefined non-primitive objects with size and position attributes, such as box, cylinder, beam, sphere, cone, etc.;

[1] Quadrilateral is a polygon with four vertices in 3D.

```
model C "using classes"
 PrimitiveGeometry x(c={Sphere(p={0,0,0},radius=1)});
 ...
end C;

model A "using annotation"
 annotation(PrimitiveGeometry({Sphere(p={0,0,0},radius=1)}))
 ...
end A;
```

Figure 1: Example of Modelica classes with graphical annotations and with graphical classes.

- surfaces defined as splines[2];
- trees with nodes for transformations, switches[3] , levels-of-detail etc.;
- visualization of forces, speed, acceleration etc., e.g. in the form of arrows;
- objects that are able to refer to graphic data in other formats (such as VRML[12] or STL[1]) and/or other files;
- objects that change their shape during dynamic visualization (non-rigid, flexible body visualization). The shape is specified by some constants and variables, and the values for the variables is computed during simulation.

Visualizations defined in Modelica models can be highly dynamic since all the numerical values used in graphic definitions can be not only constants, but arbitrary Modelica variables or numerical expressions.

3 Graphic Object Representation in Modelica Code

An important question is *how* to add geometric or graphic information to a Modelica model. These are not equations, not functions, not variables, but tree-like graphs with structural information, numerical constants and variables. The structure should match certain syntax rules to be readable by simulation and visualization software.

Two alternatives can be considered (see Figure 1):

Using classes: User-defined models of physical objects should inherit certain general graphic classes (from a new special graphic Modelica library) or include components of those classes. For instance, currently for visualization with DymoView[4] or MVISLIB[6] a special

[2]Sometimes there is a need to visualize surfaces defined via parametric functions given in a Modelica model. This case requires a very special treatment of these functions, and using function names in the same way as variables. Since this deviates too far from Modelica syntax and complicates implementation we don't cover this topic in this report.

[3]*Switch* nodes allow the user to specify several completely different graphic objects, and alternate between them depending on the switch variable value.

```
Icon(
 Rectangle(
  extent=[-90, 15; 0, -15],
  style(color=8 , gradient=2 ,
        fillColor=8 , fillPattern=1 )),
 Rectangle(
  extent=[0, 50; 100, -50],
  style(color=8 , gradient=3 ,
        fillColor=8 , fillPattern=1 )),
 Text(
  extent=[-100, 60; 100, 122],
  string="%name")),....
```

Figure 2: Example of Modelica annotation defining an icon.

class `VisualMbsObject` should be used (or classes that use or extend this class). Each class instance can include a description of some corresponding graphics.

Using annotations: The classes might contain special *annotations* which define geometric and graphic properties. Annotations in Modelica are tree-like structures with labeled branches and attribute values, e.g. as the example shown in Figure 2

Currently annotations are used for icons, diagrams (here a particular grammar for the structure of the annotations is used) and for documentation (a string with free contents).

3.1 Choice Between Classes and Annotations

The first alternative (using classes) requires creating a Modelica visualization library (with a set of Modelica classes) and adding new classes for particular geometric structures to this library. The second variant might require a specific analyzer (parser) for graphic annotations, which can be an extension to the Modelica parser.

Sometimes there is a need to include some format (e.g., VRML[12]) directly (verbatim) into the Modelica model code. In this case annotations can conveniently be used since they are effectively ignored when the model is compiled for the purpose of simulation.

Sometimes there is also a need for dynamic modification of parameters of some geometric features during simulation or visualization. For instance, the height of a box increases as a function of time; its mass and center of mass should be computed for simulation; the shape of the box should be updated for online and offline visualization [6] as well. This can be done by placing some variables into geometric classes and setting up some equations to connect these variables with the rest of the simulation. This cannot be done easily in the annotation approach, since annotations do not affect the simulation and cannot affect the visualization of simulation results.

The architecture of a special environment for handling shape annotations is described in Section 7.

It would be convenient to ignore all information related to graphics when the model is simulated just to obtain numerical values of model dynamics. In such a case classes and instances with graphic information should be "detached" from the model in order to reduce the number of classes, equations and variables. This can be needed for higher performance, memory reduction and speed-up of the Modelica compiler. It is easy to detach all the annotations at once, but it is more difficult to switch off all geometry-related classes of an application.

Each Modelica class or instance might have an annotation defining some geometric and/or graphic properties. Obviously, there can be classes that contain graphic annotations only, and these can be instantiated or extended (inherited) in other classes. Therefore both syntax alternatives have roughly the same capabilities.

There are annotations defining icon geometry in Modelica [9]. The icons can be displayed graphically in Modelica implementations[4, 7] (Figure 2). These icons are displayed as 2D graphic primitives when appearing in Modelica connection diagram editors. These annotations are not used for simulation purposes and not stored in the simulation output data.

In the second alternative the proposed graphic annotations will represent 3D graphic primitives. Some special component in the Modelica implementation should care of the task of analysis and appropriate handling of these specific annotations.

A problem arises with geometry definition in annotations since sometimes the geometry contains *connectors* (such as `MbsCutA`) and there is no place in the Modelica syntax to specify connection between a class instance inside an *annotation* with the rest of the model. However, it is possible to write a variable name in the annotation.

For describing the structure of graphics there is a need for non-homogeneous arrays, i.e. arrays consisting of instances of different classes. This is not possible in standard Modelica class syntax, but it is allowed in annotations.

Thus, both approaches have some drawbacks. The rest of this report is based on the *second* alternative (using annotations). The syntax of annotations, however, is described by Modelica class notation.

3.2 Shape Structure and Modelica Model Structure

A Modelica model refers directly and indirectly to a number of classes. These classes have components (variables) of types defined by some other classes. Instantiation of variables and equations always starts from the last (the main) class of the model (this class corresponds to the root node in the tree of Modelica classes), and continues recursively to the components.

The total collection of geometric bodies included in a Modelica model (when used for simulation or visualization) is specified recursively. The shape corresponding to a whole Modelica application is a combination of the shapes of all instances declared in the last (the main) class of the textual presentation of the application model.

If a class defines its own geometric annotations it can refer (within its attribute structures) to the annotation of the parent class (for this purpose we introduce a special keyword `Super`) and attributes of the components of the class (we introduce the keyword `Component(`*name*`)`). By default

```
model A
 annotation(PrimitiveGeometry({Sphere(p={0,0,0},radius=1)}))
end A;
model B extends A;
 Real r;
 annotation(PrimitiveGeometry({Sphere(p={0,0,0},radius=r)}))
 // This annotation overrides the annotation inherited from A
end B;
model C extends A;
 A a;
 B b;
 annotation(PrimitiveGeometry({Super,Box(...),Component(b)}))
 // This annotation overrides the annotation inherited from A.
 // C contains two spheres and a box.
end C;

model D extends A;
 A a;
 B b;
 // No annotation. D contains three spheres.
end D;

model E;
 C c; D d; // contains five spheres and a box.
end E;
```

Figure 3: Example of Modelica annotations defining geometries.

the annotation of the parent class is overridden and annotations of all components are ignored.

If a class does not define its own shape attributes, then the shape of the superclass and the shape of components are just joined together:

```
Super, Component(name1), Component(name2), ...
```

Figure 3 gives definitions of five classes with geometric information. The actual shapes associated with each class are given in the comments.

Visualization and geometric properties are inferred from the shape properties. The visualization of a complete Modelica application consists of the shapes of all the object instances. If geometric information is enough for computation of object volume, then the volume associated with a whole Modelica application is just the sum of volumes of all the instances (it is assumed that they do not penetrate each other).

4 Geometry Definition Syntax in Modelica

The library of data structures that can be used for geometric data description is presented in Figure 4.

Generally, graphics can be defined using two main approaches:

- Specification of a set of primitives (including triangles, spheres, etc.) and operations (translation, rotation, extrusion, etc.).
- Using an external graphics format (such as STL[1], VRML[12], DXF[2], etc). The names of the files that contain the graphics are specified *or*

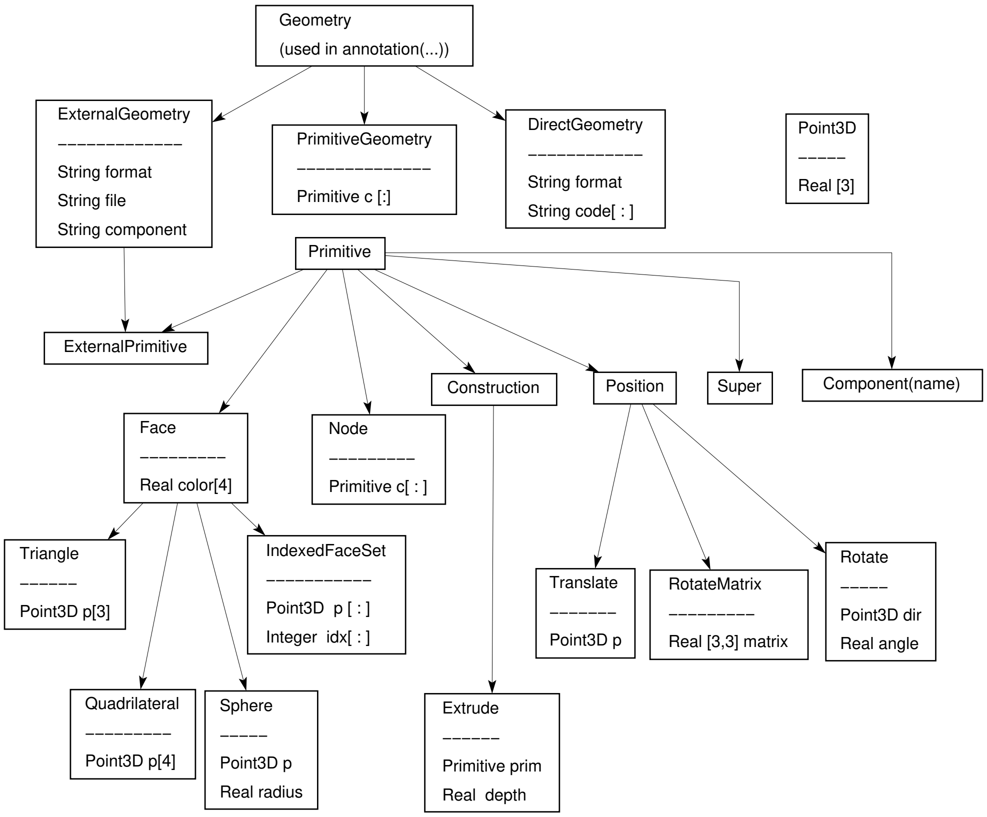

Figure 4: Relationship diagram for data structures describing geometric annotations.

the graphics in its external format is included directly (verbatim) into the model.

4.1 Syntax for External Graphic Format Specification

Some components or classes might have their graphics and geometry stored in an external graphics format.

If an external graphics format is used for a Modelica model, we need to specify the following:

- the format identifier,
- the external file name of the graphic file,
- (optional) file component identification.

In these cases an annotation is attached to a component or a class, e.g. as below:

```
annotation(ExternalGeometry(format="STL",
                            file="arm001.stl"))
```

There exist many different 3D graphics formats. The STL format is just one of the available portable formats for description of static rigid bodies.

This format may include sub-parts in the geometry description. Typically, assemblies of components are described as trees. It is possible to access subcomponents via a unique name or a sequence of names.

```
annotation(ExternalGeometry(format="STL",file="robot.stl",
                            component="17"))
```

Graphic information in an external format can be included into the Modelica model directly (verbatim), as an array of strings:

```
annotation(DirectGeometry(
    format="STL",
    code={"solid arm",
          "facet normal 0.0 -1.0 5.9e-08",
          "outer loop",
          "vertex -1.9e+01 5.1e-07 1.08e+01",
          "vertex -1.9e+001 -6.2e-007 -8.3",
               ...}))
```

5 Primitive Geometric Objects

A geometric model might contain many different primitive objects like triangles, quadrilaterals, spheres, boxes, and parametric surfaces.

In particular the following types should be available:

```
type  PrimitiveGeometry=Primitive [:];

record Primitive;
// has no specific components
end Primitive;

type Point3D = Real [3];

record Face
    extends Primitive;
    Real color[4]; /* RGBA */
    /* other features can be here */
end Face;

record Triangle
    extends  Face;
    Point3D  p[3];
end Triangle;

record Quadrilateral
    extends  Face;
     Point3D  p[4];
end Quadrilateral

record Sphere
     extends  Face;
     Point3D  p;
     Real radius;
end  Sphere

record IndexedFaceSet
     extends  Face;
     Point3D  p[:];
     Integer idx[:];
```

An annotation defining a geometry might look like:

```
annotation(
 PrimitiveGeometry({
    Triangle(p={{0.3, 0.4, 0.2},
                     {1.4, 0.4, 0.2},
                     {1.4, 0.5, 0.3}}),
    Sphere(p={0.2,0.4,0.6}, radius=1.2  )
  }))
```

6 Position Specification

Generally geometry is specified as a tree. The semantics of `Translate`, `Rotate` and `Scale` re the same as in OpenGL[11]. Other OpenGL-like constructs can be added in the same way. These primitives define transforms from an old coordinate system to the new coordinate system, and all further drawing in the current list of primitives is applied in the new coordinate system.

Corresponding definitions are:

```
record Position extends Primitive
end Position;

record  Translate
    extends Position;
    Point3D  point;
end  Translate;

record  Rotate
    extends Position;
    Point3D  direction;
    Real     angle;
end  Translate;

record  RotateMatrix
    extends Position;
    Real [3,3]  matrix;
end  Translate;

record Scale
   extends Position;
   Point3D size;
end Scale;
```

A `Node` is a `Primitive` that might contain a number of other nodes:

```
record Node
    extends Primitive; /* it is a primitive */
    Primitive [ : ];
     /* contains some other nodes and other primitives */
end Node
```

The position specification is stored in the tree, together with the faces. The example below defines two spheres. As result of combination of two translations, the center of one of them is shifted to (1,0,1), the other becomes (1,0,-1) .

```
annotation(
  PrimitiveGeometry( {
     Translate( point={1,0,0}),  // Translates both nodes
     Node({
        Translate( point={0,0,1}), // Translates only one sphere
        Sphere( point={0,0,0}, radius=1)
               })
              ,
       Node({
        Translate( point={0,0,-1}), // Translates only one sphere
        Sphere( point={0,0,0}, radius=1)
                })
     }))
```

6.1 Specification of Level of Detail

In case of rendering we can consider specification of levels of detail (LOD). The primitive list can differ depending on the LOD required. Different primitive lists will be chosen from a particular branch, depending on the current detail level. Different LOD levels can be specified for the attribute and for the class.

There can be a `Switch` nodes that selects a particular child node (from several alternative nodes) according to some parameter or variable. For example, the variable p can only take the value 1 (a sphere will be seen) or 2 (a box will be seen).

```
annotation(
 PrimitiveGeometry(
    Switch(
      name=p ,
      alternatives={
          Sphere( p={0,0,0}, radius=1.0 ),
          Box ( p1={-1,-1,-1}, p2={0,0,0})
          })))
```

There is a similar `Switch` construct in the VRML language.

7 Implementation Outline

In this section we present an outline of a future implementation of environment components necessary for integration of 3D geometry and 3D graphics with Modelica.

The structure of this implementation is shown in Figure 5.

Usually the Modelica model is supplied to the compiler, and all the annotations are ignored. In order to extract necessary information a special *pre-compiler* should be designed. It takes Modelica model with 3D geometry and graphical annotations and transforms it to labeled tree representation (LTR). The labels at the top level of the tree correspond to the variable paths of the variables containing some graphical annotations. For instance. model `E` in Figure 3 corresponds to the tree in Figure 6:

The labeled tree representation (LTR) should be supported as a tree in memory. Additionally, it should be possible to write and read such structures to and from a file.

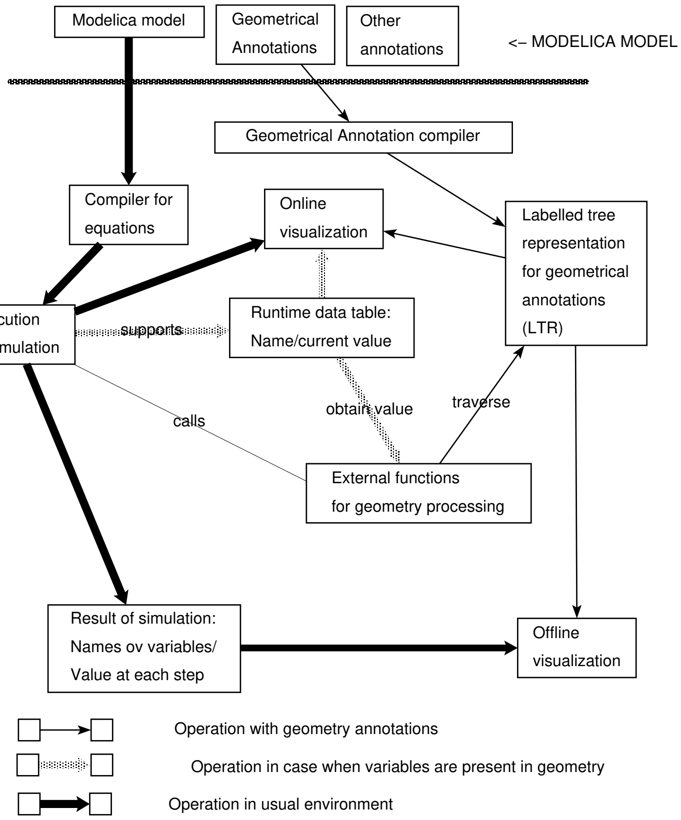

Figure 5: Implementation structure for Modelica environment working with graphical annotations.

When a simulation is running, external functions can be called in order to obtain some geometry data. For instance, the current volume of some body, or collision force that currently occur between the bodies. These functions (e.g. `GetVolume(‘‘d’’)`) find the corresponding branch or set of branches in the LTR. If some variables (such as `d.b.r`) are needed during the computations, the values can be obtained from the runtime data table. This table contains the names of all available variables and their current values.

Some functions (e.g. to find the volume of a rigid body, which never changes during simulation) should be called just once, before the actual simulation starts. The result of these functions can be used in the same way as `parameters` in Modelica.

The same LTR is used when the geometry is required during visualization. Necessary variable values are fetched from the file with simulation results.

8 Conclusions

In this report a flexible notation for geometry description is discussed. In order to obtain the necessary flexibility and short notation, some devia-

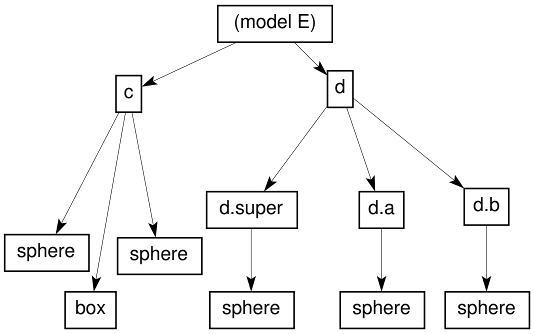

Figure 6: Labeled tree representation for geometric annotations.

tion from the standard Modelica syntax is necessary. This is one of the reasons in using annotations instead of classes where a new standard can easily by defined and old standards can be extended. The proposed standard requires implementation of some extra tools, including an annotation translator. These tools do not break the current implementation of Modelica [4, 7] and furthermore require no essential changes in it. Therefore it can be considered as a pure Modelica extension.

References

[1] 3D Systems, *Stereo Lithography Interface Specification*, 3D Systems, Inc., Valencia, CA 91355. Available via http://www.vr.clemson.edu/credo/rp.html.

[2] AutoDesk, *AutoCad documentation. DXF Format*, http://www.autodesk.com

[3] Cycore AB, *Cult 3D Home Page*, http://www.cult3d.com

[4] Hilding Elmqvist, Dag Brück, Martin Otter, *Dymola, Dynamic Modeling Laboratory, User's Manual, Version 4.0*, from Dynasim AB, Research Park Ideon, Lund, Sweden, http://www.dynasim.se

[5] Emacs, http://www.emacs.org

[6] Vadim Engelson, *Tools for Design, Interactive Simulation and Visualization for Dynamic Analysis of Mechanical Models*. Linköping Electronic Articles in Computer and Information Science, ISSN 1401-9841, Vol. 5 (2000): nr 007. Available at: http://www.ep.liu.se/ea/cis/2000/007/

[7] MathCore AB, *MathModelica* , software tool for modeling, simulation, and visualization. Available from MathCore AB, http://www.mathcore.com.

[8] Modelica Design Group, *Modelica WWW site*, http://www.modelica.org

[9] Modelica Design Group, *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling. Tutorial and Rationale. Version 1.3* December 15, 1999. Available via http://www.modelica.org

[10] Martin Otter, Hilding Elmqvist and François E. Cellier, Modeling of Multibody Systems with the Object-Oriented Modeling Language Dymola, *Nonlinear Dynamics*, 9:91-112, 1996, Kluwer Academic Publishers.

[11] SGI, *OpenGL Web Site*, http://reality.sgi.com/opengl/

[12] VRML Consortium, *VRML Repository*, http://www.sdsc.edu/vrml/