Modelica3D - Platform Independent Simulation Visualization

Christoph Höger[1], Alexandra Mehlhase[1], Christoph Nytsch-Geusen[2], Karsten Isakovic[3], and Rick Kubiak[3]

[1]*Technische Universität Berlin*
[2]*Universität der Künste Berlin*
[3]*Fraunhofer FIRST*

Abstract

Modelica3D is a platform-independent, free Modelica library for 3D visualization. Its implementation is based on a message-passing architecture. Through its loosely-coupled architecture, Modelica3D can be combined with different rendering-tools. It is also highly extensible and scalable.

Keywords: 3D Graphics, Library, Platform Independence, Free Software, Structural Dynamics, Loose Coupling, Message Passing

1 Introduction

Simulation results in Modelica are usually visualized using two-dimensional plots. System states are shown as functions over time or each other. While this is of course a very natural way to approach the presentation of simulation results, it is not sufficient in some aspects:

In Modelica, simulation-models are composed of reusable software fragments. Thus an interesting quantity might not be present directly, but in form of a relation between multiple system states (e.g. distances between different objects). The solution is to either change the model, post-process the simulation results, or to switch to a more complex visualization method.

Additionally, a simulation might be used to drive interactive real-time simulators (e.g. for training purposes) or to present certain facts to a non-simulation audience. In both cases the usage of 3D graphics might lower the barrier significantly for team members, which are not as familiar with the simulation as the responsible engineer. Therefore it is important that the visualization aspects can be controlled from within the simulation environment. On the other hand visualization experts (and expert-tools) are necessary to create realistic and usable 3D-graphics. The Modelica3D library aims to provide a solution for these requirements.

1.1 Contribution

In this paper we will demonstrate how a visualization library (called Modelica3D) can be implemented by using only standard Modelica features. The library itself is available under a free software license. We will show how a tight coupling between the library and the rendering-tool can be avoided. This loose coupling allows the visualization of structural dynamic systems. Additionally, we show how the underlying message-exchange API makes the Modelica3D API both flexible and extensible. Finally, an example is given, demonstrating the scalability of Modelica3D to industrial models.

Finally, the method proposed here is not limited to 3D-graphics. The same means could be used to control e.g. sound output or any other simulation feedback. In that way, Modelica3D demonstrates how simulations can control *effects* beyond their simulation environment.

The rest of the paper is organized as follows: First, we will discuss the state-of-the-art of 3D visualization in the Modelica ecosystem. Then, a technical overview over Modelica3D's architecture is presented. This includes a discussion of the overall design as well as solutions to overcome some Modelica-specific limitations. Second, we will show how Modelica3D can be used to simulate an existing library (Modelica.MultiBody) to achieve tool-independent state-of-the-art visualization. As a second use-case a recently developed technique for finite-state structurally dynamic systems is extended with Modelica3D visual-

ization. Finally, we will evaluate the library by implementing a large-scale industrial model visualization.

2 State of the art

During the last 10 years different approaches were tested to integrate scene descriptions of 3D bodies in the Modelica language or to support 3D visualizations by the Modelica simulation tools.

The first fundamental analysis and conceptual work in this field was done by Engelson [3]. Two alternative ways were discussed for the integration of 3D object information in Modelica: First, the definition of a basic set of "graphical" Modelica classes, which make a representation of primitive 3D objects (e.g. triangle, sphere) and position operations with this objects (e.g. translation, rotation) in user defined physical models possible. Second, the direct integration of the 3D object information as "graphical annotations" into the physical models self.

Another approach of a annotation concept for the embedding of 3D geometries in Modelica was developed from [5], where specialized 3D annotations for model classes and objects and a standardized description of 3D geometries and the related body topologies (in this case the X3D standard) were combined. Within the tool specific approaches, individual ways for the 3D information integration were done by the software developers. The greatest disadvantage consists in the incompatibility of the 3D models, caused by the use of vendor specific 3D information.

The simulation tool SimulationX from ITI supports both for his own Modelica libraries and also for user written Modelica libraries the visualization and animation of 3D objects. With the help of an 3D editor tool, the 3D information is stored in the physical Modelica models and also in related non standardized annotations. The 3D editor supports the definition of simple and complex bodies, which are constructed by the combination of standard 3D primitives and also specialized objects such as gears and spiral springs.

The simulation tool Dymola from Dassault Systemes supports for selected Modelica libraries the visualization and animation of 3D-objects, mainly for the MultiBody-Library. For this, specialized visualization classes for 3D primitives were introduced. Further, complex 3D geometries, based on external definitions of 3D-shapes via dxf-files are utilized. The MultiBody package of the Modelica Standard Library uses data structures defined in Modelica.Services to calculate a *complete* continuous time model of the 3D visualization geometry. This approach does not allow for *effect-events* (e.g. deformations, material changes).

The visualization framework SimVis for 3D modelling and simulation with Modelica was developed by the German DLR [2]. On the modelling side, a new developed ExternalDevices-library represents the base for the 3D visualization and interactivity. For the simulation experiments three different types of input devices (keyboard, joystick, 3D space mouse) supports the direct 3D interaction by the user. The technical base on SimVis is OPENGL and OPENSceneGraph. Different use cases were analysed within SimVis such as flexible body simulation, energy flow simulation, Head-Up-Display simulations, hybrid cars and robot simulation.

3 Modelica3D

In this section we discuss the architecture of Modelica3D and the design decisions that lead it. As Modelica3D is a purely non-physical library, there are no modeling concerns (e.g. reusable and understandable components) that need to be addressed. Instead, Modelica3D focuses solely on effects *outside* of the simulation. Thus, we could focus on general software design principles and the goals motivated earlier.

3.1 Design Decisions

First of all, Modelica3D should be platform independent: Only methods that are part of the Modelica Specification [1] should be used. This rules out the development in form of an extension to an existing platform and a solution based on vendor-specfic annotations. Any tool that follows the specification should be able to use Modelica3D directly. During the development we used OpenModelica [4].

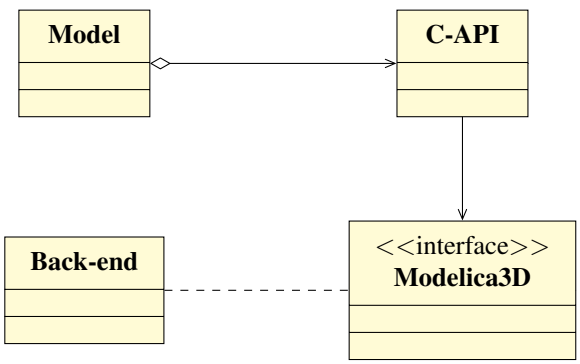

Figure 1: Modelica3D architecture

Instead, Modelica3D must be shippable as a library.

DOI 10.3384/ecp12076485

This library must contain a layer of Modelica-Code, which allows access to the 3D API from any model. On the other hand, extensions which cannot be expressed in Modelica need to be implemented in a language that is supported through Modelica's external function interface. Since only C and Fortran are currently specified, C is a natural choice, being the "lingua franca" of platform independent development.

The second important requirement was loose coupling between front-end (the simulated model) and back-end (the rendering-tool). While with the choice of C as implementation language, several options for accessing rendering-tools exist, directly linking the back-end would cause several drawbacks:

- Only few back-ends can be used as a library. Even if they do (e.g. OpenSceneGraph), the viewer usually requires a lot of additional features (user interface, inputs, file management). Providing those features to a Modelica model in a platform independent implementation would require lots of additional work for each back-end and thus only allow very few implementations.
- A fixed C API would not only put an additional burden on developers who want to extend the library. It would also of hinder the maintenance, since every back-end would effectively require it's own C-library (including it's own bugs). Ideally the parts written in C should be as small as possible instead, leaving the lion's share of work to Modelica and back-end experts.
- Linking works only locally. In times of distributed computing it seems unreasonable to demand simulations running on the same physical machine as visualization.

So instead of directly linking 3D API functions into a Modelica model, we chose to use interprocess communication (IPC). That way, front-end as well as back-end can run as dedicated processes while sending respectively receiving messages. Any back-end needs to implement a common *interface* (which is simply the set of messages accepted).

Because visualization should not influence the simulation results, the communication between front- and back-end is *unidirectional*. In our design this allows a further simplification of the message-exchange protocol: Since the front-end does not expect any messages from the back-end, the communication can work *synchronously*. This also fits into the event-driven modeling style of Modelica. Note, that by using *time-events* for the event handling, the effect on the simulation performance can be minimized by the simulation tool, as discussed e.g. in [8].

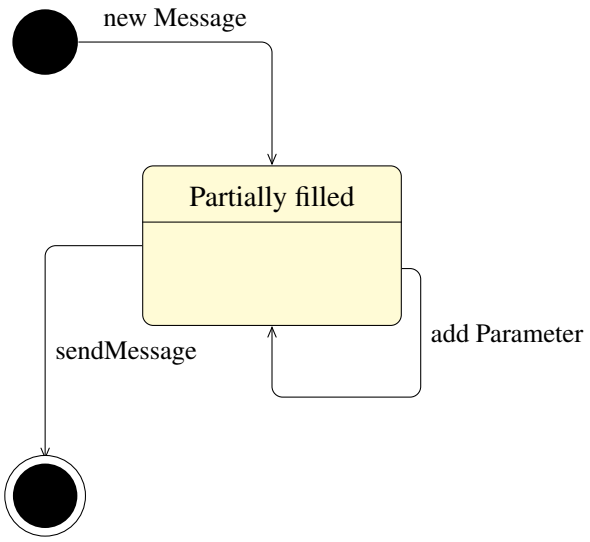

Figure 2: modbus message lifecycle

3.2 Implementation

With the design decisions settled, the first task was to implement an *extensible*, *synchronous*, *platform-independent* IPC layer in Modelica. Instead of reinventing the wheel, we chose to use an existing IPC solution and wrap around it's C-interface. Because of it's availability, maturity, and simple C-API, the choice fell on dbus, the current de-facto standard for IPC on Linux [7] [1].

The first part of Modelica3D is thus a thin Modelica wrapper around dbus, called *modbus*. Modbus allows creation and sending of arbitrary messages as *External Objects*. Message objects can be allocated, equipped with parameters and send over a *connection*.

Since modbus only uses very few of features of dbus (only one-to-one communication, uniform, statically known messages etc.), this implementation could be considered overhead and in a sense it certainly is. On the other hand, the implementation itself becomes rather simple: Currently it consists of 96 lines of Modelica- and 216 lines of C-Code. In case a faster solution is needed, modbus should be trivial to port to whatever IPC-mechanism seems appropriate. Additionally, this strategy makes it unnecessary to store and continuously calculate the 3d geometry. Depending on the scene, this might yield significant runtime and

[1] It has been ported to windows, too.

memory improvements over methods like the MultiBody visualization.

To further reduce the size of the interface (and make it more convenient to implement), all methods provided by the Modelica3D API accept *named* parameters and allow to emit some of them (using defaults instead). Internally modbus implements this by storing the parameters in a dbus map-object. This yields some further overhead (dbus' internal type-checking becomes quite useless), but allows more selective updates on graphical objects (e.g. it would be possible to only change the Z-axis location of an object without even knowing its X- and Y-axis locations).

Method name	Description
`loadSceneFromFile`	Loads a complete scene from a file
`createMaterial`	Create a material primitive
`applyMaterial`	Use a material on an object
`createBox`	Create a box primitive
`createBoxAt`	Create a box primitive, with a given orientation
`createSphere`	Create a sphere primitive
`createCylinder`	Create a cylinder primitive
`createCylinderAt`	Create a cylinder primitive, with a given orientation
`createCone`	Create a cone primitive
`createConeAt`	Create a cone primitive, with a given orientation

Table 1: Modelica3D setup-methods

3.3 Alternatives

As already mentioned, the choice of dbus for message exchange was mainly due to pragmatic reasons. Any other platform-independent IPC solution might suffice as well. Albeit, there is a fundamentally different design that needs some discussion. In certain settings, *every* message exchange, no matter how lightweight, may cause a too big delay:

Consider a real-time system running at 60 or more fps and visualizing large sets of objects (e.g. a scene in a game engine). Since synchronous message exchange requires at least 2 context switches, and a context switch is rather costly [6], we can estimate a theoretical upper limit of 10^5 simultaneously animated objects. Any practical limit will of course be much smaller, since not only context switches are required. Basically, this means, that, independent of the visualization or simulation complexity, a system composed of some thousands of objects that shall be visualized, cannot be rendered in real-time, when message-passing is used.

So instead of sending lots of small messages about the state of each object, front- and back-end could use *shared memory* to exchange large chunks of data very fast. Unfortunately, such a solution is hardly platform independent and more complicated to implement (since both sides would need to synchronize their access on that data). But since it is obviously a useful design alternative, further research seems to be appropriate.

3.4 Data structures and operations

Modelica3D comes only with a very small set of data structures. Next to the already mentioned modbus objects, it provides a system state record, a controller model and a definition of object-ids. The system state basically only combines a modbus context object with a connection and a counter for the current frame. The controller in turn wraps the state and provides a sampled boolean signal depending on a selected framerate. It also modifies the state's frame counter according to the current `time` and can send a `stop`-message to the back-end at the end of simulation time.

Method name	Description
`rotate`	Change an object's orientation
`moveTo`	Change an object's location
`moveZ`	Move along the Z-axis only
`scale`	Change the size of an object
`scaleZ`	Scale along the Z-axis only
`setAmbientColor`	Sets the ambient color value of a material
`setDiffuseColor`	Sets the diffuse color value of a material
`setSpecularColor`	Sets the specular color value of a material
`setMatProperty`	Changes a given (named) material property

Table 2: Modelica3D modification-methods

Id-objects are currently only heap-allocated strings. But on demand, they might be easily exchanged with a more complex internal implementation (e.g. if the library would want to implement hashing or collect statistics on the objects).

DOI 10.3384/ecp12076485

Most methods in the Modelica3D API fall into two distinct groups: There are operations that describe the *setup* of a scene (table 1) and operations that *modify* a scene dynamically (table 2). The difference between them is that the latter ones need a frame number, which works as a logical clock that describes, *when* such a modification takes effect, while the former ones are always interpreted once at the beginning of the animation. The only exception from that pattern is the `stop`-operation. Sending this message tells the client to stop listening for further messages.

The set of currently implemented operations is rather small. But due to the design of Modelica3D, additional operations might be added by simply extending the package (and at least one backend). No recompilation of the C-library is required.

Listing 1: `moveTo`-method in Modelica

```
function moveTo
  input State state;
  input Id id;
  input Real p[3];
  input Integer frame=state.frame;
  output String r;

  protected
  Message msg = Message(TARGET,
   OBJECT, INTERFACE, "move_to");
  algorithm
    addString(msg, "reference",
      getString(id));
    addReal(msg, "x", p[1]);
    addReal(msg, "y", p[2]);
    addReal(msg, "z", p[3]);
    addInteger(msg, "frame", frame);
    r := sendMessage(state.conn, msg);
end moveTo;
```

Implementing an operation in Modelica is not difficult. Listing 1 shows the `moveTo` function is implemented. It consists of allocating a message object (from the dbus-connection constants for the target and the dbus-interface and the method's name), adding parameters to that message, and finally sending it. Further operations should follow that pattern.

3.5 Back-ends

Currently, Modelica3D contains two back-end implementations. They demonstrate two distinct kinds of visualization tools. The first tool, blender [11], is a 3D-modeling tool which can render high-quality movies. Blender provides a python interpreter for scripting purposes. Thus it was a natural choice to implement the back-end parts in python.

Listing 2: `moveTo`-method in blender

```
@mod3D_api(reference = defined_object,
           frame = positive_int)
def move_to(self, reference,
            x=None, y=None, z=None,
            frame=1, immediate=False):
  o = data.objects[reference]
  context.scene.frame_set(frame=frame)
  if immediate:
    o.keyframe_insert('location',
                      frame=frame - 1)

  if (x != None):
    o.location.x = x
  if (y != None):
    o.location.y = y
  if (z != None):
    o.location.z = z

  o.keyframe_insert('location',
                    frame=frame)

  return reference
```

Listing 2 shows the implementation of the `moveTo`-method in the blender back-end. The `mod3D_api`-decorator is responsible for lifting a python function into a dbus-method. That lifting is (due to the uniform signature) the same for all back-end methods. Additionally certain runtime checks might be added (e.g. checking if a given object-reference actually exists, a number is positive etc.).

Since blender provides access to it's internal data representation (`data.objects`), the rest of the method is straight-forward. It directly changes the object's coordinates (if provided by the client) and inserts an animation key-frame (allowing for interpolated movement, if necessary).

The other back-end was implemented using OpenSceneGraph [9], a free 3D-engine. Unlike blender, it does not provide modeling facilities. Instead, its scope is fast, real-time rendering. That way we demonstrate how Modelica3D might also be used in interactive applications[2].

4 Usage

In this section we show, how Modelica3D can be used to visualize different kinds of simulations. First, we will show how Modelica3D can handle state-of-the-art visualizations on the basis of the MultiBody library. Second we will describe the visualization of a simple, structurally dynamic system.

[2]The graphical output, the input needs to be handled with some other tool or library.

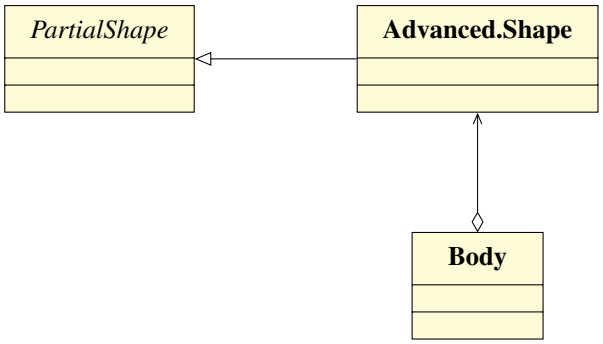

Figure 3: MultiBody visualization-class structure

4.1 Visualizing MultiBody

As mentioned earlier a popular (if not the most popular) method of visualization comes with the Modelica Standard Library: All models from the MultiBody library can be visualized according to their geometrical structure. Since a 3D-mechanical library naturally contains information about the location and relative rotation of objects, visualization is straight-forward.

This makes the MultiBody library a good example of how Modelica3D can be used in such existing complex hierarchies. In this use-case, all visualization information culminates in one class, the `PartialShape` (Figure 3). That class basically consists of the shape parameters (length, material etc.) and a translation matrix. This gives us an insertion point of where to insert the Modelica3D functionality. In a first step, we introduced a controller object into this class, to hold the Modelica3D context information. Since this controller needs to be unique among all shapes, it is naturally marked as **outer** (Listing 3).

Listing 3: Additional fields of PartialShape

```
outer M3D.Controller m3d_control;
Id id;
Id mat;
String res;
discrete Real[3] pos;
modcount.Context initContext
  = modcount.Context();
```

Additionally an object-id is added for both the shape's material and geometry. The variable `pos` holds the current position (resolved from the translation matrix), while `res` captures the result of each operation (ensuring that they are evaluated at least once). Finally a modcount-context object is used to ensure singleton evaluation of message generation. With those fields present, the animation dynamics can easily be implemented by **when**-algorithms:

Listing 4: PartialShape algorithmic dynamics

```
when initial() and
  modcount.get(initContext) <> 1 then
  id := shapeDescrTo3D(m3d_control.state,
    shapeType, length, width, height,
    lengthDirection);
  mat := M3D.createMaterial
    (m3d_control.state);
  M3D.setAmbientColor(m3d_control.state,
    mat, color[1] / 255, color[2] / 255,
    color[3] / 255, 1.0,  0);
  M3D.setSpecularColor(m3d_control.state,
    mat,
    specularCoefficient * color[1] / 255,
    specularCoefficient * color[2] / 255,
    specularCoefficient * color[3] / 255,
    1.0, 0);
  M3D.applyMaterial(m3d_control.state,
    id, mat);
  modcount.set(initContext, 1);
end when;

when m3d_control.send and
  modcount.get(initContext) == 1 then

  pos := r + Frames.resolve1(R, r_shape);
  res := M3D.rotate(m3d_control.state,
    id, R.T, m3d_control.state.frame);
  res := M3D.moveTo(m3d_control.state,
    id, pos, m3d_control.state.frame);
end when;
```

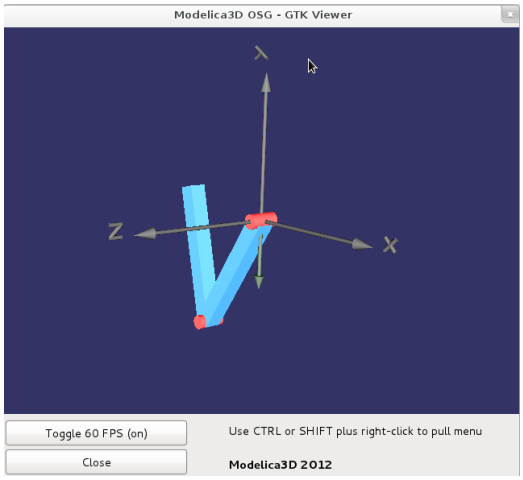

Figure 4: MultiBody visualization with Gtk+/OpenSceneGraph back-end

In this example we omitted some dynamics like changing lengths or colors, since this is unused in ourexample models (all shapes basically remain constant during simulation). If necessary, those details can be added here easily. Also messages are always sent, even if there is no movement on every frame. A more sophisticated implementation could detect a relevant

DOI 10.3384/ecp12076485

change in the model and decide whether or not to update the visualization state.

With this small extension, we were able to simulate and visualize the examples from the standard library with the OpenModelica simulation tools (Figure 4). Thus we successfully demonstrated that by only using standardized techniques, we could visualize complex models.

4.2 Variable-structure modeling

A variable-structure model is a model which can consist of different systems of equations (with different numbers of equations) and different variables depending on the simulation time. This is of interest, when a model has different levels of detail. Another application is to change the model's behavior described by a different set of equations.

Simulation engines like Dymola, SimulationX and OpenModelica do not support such changes. To overcome this drawback and still be able to use common simulation engines for the simulation of a model, a Python framework was introduced in [10]. This framework allows the user to specify a variable-structure model. The user can specify an arbitrary number of models and switches between these models. The user also has to specify how the new model should be initialized with the end values of the old mode.

For now the simulation engines Dymola, OpenModelica and Simulink are integrated in the framework. But the framework is implemented in such a way that other environments can be added quite simply. After specifying the model the Framework starts to simulate the first model in the chosen simulation environment. The model needs a stop condition which specifies when another model should be used and defines the next model. The framework uses this information to switch to the next model and initialize this model with the correct values.

We demonstrate this approach with a simple bouncing ball model. This model could of course be modeled without the variable-structure approach, but it is used for didactic purposes for the modes are easy to understand and the results are good to visualize.

This model consists of two separate modes. The first is the common falling mass model which is valid as long as the ball does not touch the ground. As soon as it touches the ground the ball is modeled as a spring/damper system and therefore the elastic deformation of the model and the bouncing back off the ground can be modeled easily. As soon as the ball leaves the ground again the falling mass model is used again. Figure 5 shows a statechart with the two modes and the switching condition is presented.

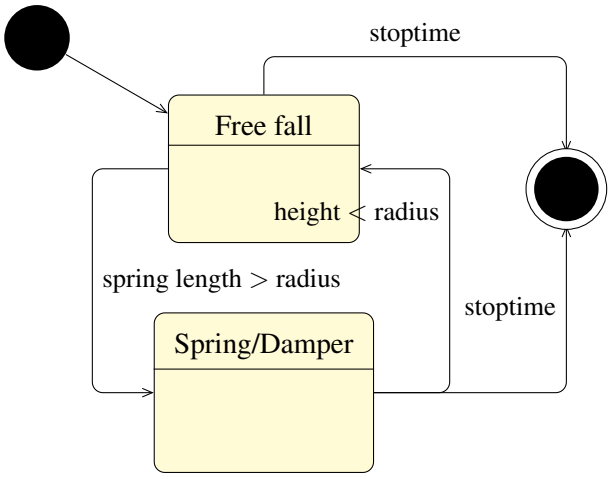

Figure 5: Statechart of the bouncing ball variable-structure model

Simulating this model with the framework and plotting the center of the ball results in the plot shown in figure 6. Here it can be seen, that the center point of the ball reaches below the radius (1.0) of the ball. This effect is caused by the elasticity of the ball in the spring/-damper mode.

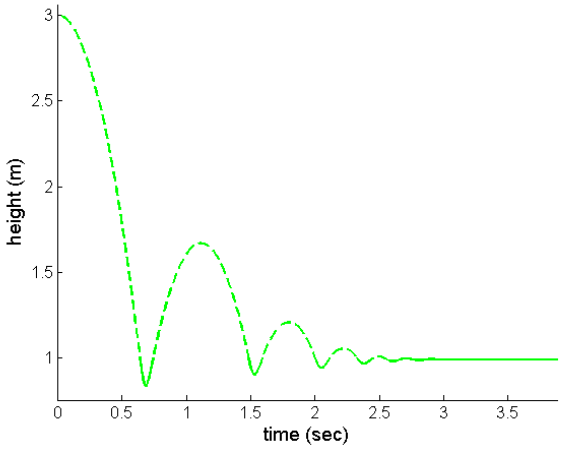

Figure 6: Center point of the bouncing ball model

A simulation of a variable-structure model with the Python framework starts simulations of the different modes sequentially. To be able to visualize such simulation results using Modelica3D, the models describing the states of the system need to fulfill two requirements:

First, they need to work on a *common* scene. Setting up such a scene is trivial: Either by directly loading it into the rendering tool at start or by creating a

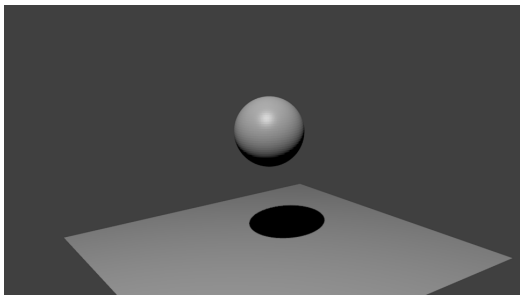

Figure 7: Ball in free-fall mode

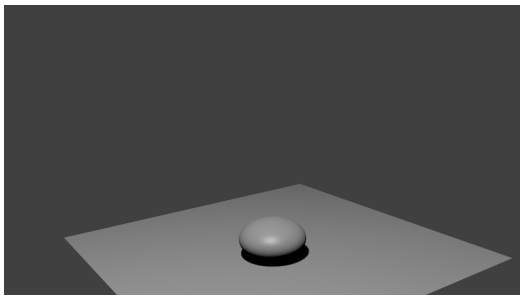

Figure 8: Ball compressed

dedicated initial state that handles all setup commands from the Modelica3D API. Here the decision of using IPC instead of direct linking pays off: Since the rendering tool runs only once, no special treatment for structurally dynamic systems is necessary. Second, all models need to know which parts of the scene they modify. In our example, both models need to know the name of the ball. This *visualization interface* cannot be statically checked.

Listing 5: Free-fall visualization

```
algorithm
  when initial() and
    modcount.get(initContext) <> 1 then
    ball := M3D.objectId("Ball");
    modcount.set(initContext, 1);
  end when;
  when m3d_control.send and
    modcount.get(initContext) == 1 then
    M3D.moveZ(m3d_control.state,
     ball, h, m3d_control.state.frame);
  end when;
```

In our example, we took a simple approach to modeling: The scene consists only of a plane representing the ground and a sphere for the ball. Camera and some lighting is added by blender. In free-fall mode, the only thing to change is the location of the sphere on the Z-axis (Listing 5). On-ground, we model the compression of the ball by scaling and moving the sphere along the Z-axis.

Since the python framework controls the activation and deactivation of the states (by extending the physical models with terminal-conditions etc.), this approach works seamlessly: Figure 7 shows the ball falling towards the plane. The compression is captured in figure 8. Naturally, the true visual effect of bouncing can not be shown in single images, but only when viewing the whole animation.

5 Evaluation

We evaluated a development version of Modelica3D (enhanced with the ability to group objects on the back-end for simpler handling of complex scenes) in a case study of a solar-thermal hydraulic system, which is integrated in the structure of a building envelope. For this objective, several sub-steps had to be realized.

5.1 Modelica3D extensions of the physical models

First, the component models of the library BuildingSystems [3] were extended with the ability to have a representation within a 3D scene and to show values such as temperatures, pressures or mass flow rates. Figure 9 shows this extension procedure for the example of a 1D-segmented thermal hydraulic model of a tube. The new model class PipeStraightVis3D was derived from the existing physical model class PipeStraight and from a general model class for 3D representation ModelVis3D.

The model extension comprises the definition of the shape of the 3D sub-primitives (here the cylinder pieces of the segmented fluid volume), the combination of them in a common container, the definition of the material (the appearance in the 3D scene) incl. the link to the sub-primitives, the alignment and merging of the sub-primitives to the common 3D representation and the mapping of the physical values to a graphical representation within the 3D model (in this case the fluid temperature of each fluid segment).

On a next level, several 3D-extended tube models and a 3D-extended pump model were combined to a simple thermal hydraulic loop. Figure 10 shows the 2D diagram of the Modelica system model on the left and the corresponding 3D animated scene on the right.

[3] http://www.modelica-buildingsystems.de

DOI 10.3384/ecp12076485

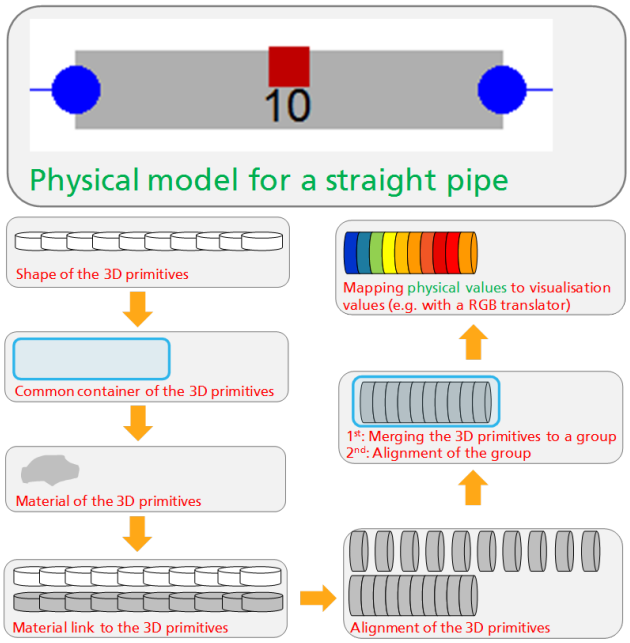

Figure 9: Extending a physical model for the use in Modelica3D

5.2 Case study of a solar thermal system

As the first complex application of the 3D visualization method, a solar thermal system for warm water production was used (Figure 10 left). The components of the solar thermal system are two evacuated tube collectors with a total aperture area of 6.34 m2 and a hot water storage with a volume of 400 liters. The roof collector is aligned to the south and tilted with an angle of 30°. An external plate heat exchanger transfers the produced thermal energy from the solar loop to the storage loop. With the help of a two-point-controller the solar pump and the storage pump are switched on,

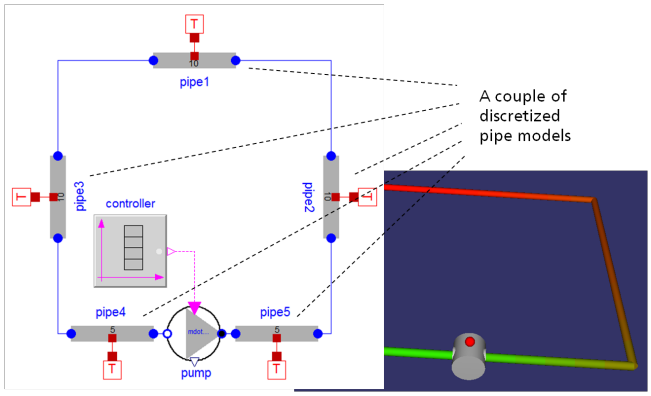

Figure 10: 2D and 3D representation of a thermal hydraulic loop

if the collector outlet temperature is 4K higher than the temperature in the lower part of the storage. As climate boundary conditions weather data from Hamburg (Germany) were used.

In the simulation scenario a load process for the thermal water storage over a time period of 24 h during a summer day were calculated. At the beginning of the load process the fluid temperatures in the collector, in the pipes and in the storage was set to 20 °C. Figure 10 (right) shows the described solar thermal system as a graphical 2D diagram, based on the "3D-extended"-components of the BuildingSystem-library.

Figure 11 illustrates the simulated transient load process for the summer day, described by the most important system variables such as the solar irradiation on the collector, the mass flow rate of the storage pump, the collector outlet temperature and the storage temperature at the bottom.

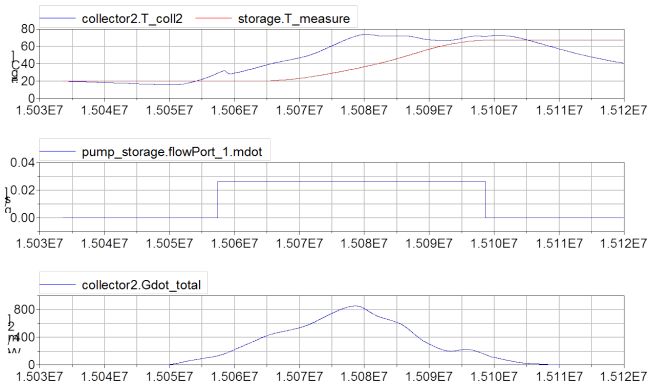

Figure 11: Simulated load process of the solar thermal system

For a clear representation within a 3D scene, the model of the solar thermal system was embedded in the 3D model of a building envelope. The 3D building envelope was modeled as a pure geometrical representation without any physical behavior. In this manner, realistic geometries and positions of different technical components of the solar thermal system (tube lengths and diameters, the required space of the storage and the collectors etc.) can be visualized. Figure 12 shows a snapshot of a the visualized transient load process of the storage during the hours before noon during a summer day in Hamburg. The different colors illustrate the temperatures of the fluid within the collector model, the tubes and the warm water storage from cold (blue) to warm (green). Because the collectors are serial connected and the cold fluid enters at first the left collector, the temperature gradient within the segmented collector model increases from left two right.

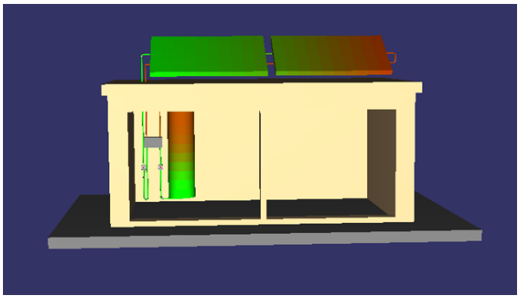

Figure 12: 3D-scene of the solar thermal system

5.3 Results

The result of the evalution can be summarized as follows:

- The developed method allows a performant representation of 3D scenes with a large quantity of animated graphical 3D elements.
- It is possible to represent complex 3D scenes with the unchanged Modelica code in different 3D environments (eg. Blender and OpenSceneGraph)
- A 3D modeling editor for a time efficient and correct configuration of complex 3D Modelica scenes is absolutely necessary

6 Conclusion

Modelica can be extended too support 3D-visualization of experiments. That extension can completely be implemented in form of a library by only using already standardized techniques. By choosing a loosely coupled, distributed architecture, the extension can support different back-ends and itself be extended easily. Additionally, innovative use-cases as variable-structure modeling are supported by this approach.

6.1 Obtaining Modelica3D

A public version of Modelica3D can is published under the terms of the GNU General Public License. The project page can be found at https://mlcontrol.uebb.tu-berlin.de/redmine/projects/modelica3d-public.

References

[1] Modelica - a unified object-oriented language for physical systems modeling, 2010.

[2] T. Bellmann. Interactive Simulations and Advanced Visualization with Modelica. In *Proceedings of the 7th Modelica Conference, Como, Italy*, 2009.

[3] V. Engelson. 3D Graphics and Modelica-an integrated approach. *Linköping Electronic Articles in Computer and Information Science. Linköping universitet*, 2000.

[4] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The openmodelica modeling, simulation, and development environment. In *Proceedings of the 46th Conference on Simulation and Modeling*, pages 83–90, 2005.

[5] Thomas Hoeft and Christoph Nytsch-Geusen. Design and validation of an annotation-concept for the representation of 3d-geometries in modelica. In *Proceedings of the 6th International Modelica Conference*, 2008.

[6] C. Li, C. Ding, and K. Shen. Quantifying The Cost of Context Switch. In *Proceedings of the 2007 workshop on Experimental computer science*, page 2. ACM, 2007.

[7] R. Love. Get on the D-BUS. *Linux Journal*, 2005(130):3, 2005.

[8] H. Lundvall, P. Fritzson, and B. Bachmann. Event handling in the openmodelica compiler and runtime system. Technical report, Technical Report 2, Dept. Computer and Information Science, Linköping Univ, 2008.

[9] Paul Martz. *OpenSceneGraph Quick Start Guide*, 2007.

[10] A. Mehlhase. A Python Package for Simulating Variable-Structure Models with Dymola. submitted, feb 2012.

[11] Ton Roosendaal and Stefano Selleri. *The Official Blender 2.3 Guide: Free 3D Creation Suite for Modeling, Animation, and Rendering*. No Starch Press, June 2004.

DOI 10.3384/ecp12076485