The DLR Visualization Library

Recent development and applications

Matthias Hellerer Tobias Bellmann Florian Schlegel
German Aerospace Center (DLR), Institute of System Dynamics and Control
82234 Wessling, Germany

Abstract

Modelica models are mathematical descriptions and therefore their simulation output typically shows numerical variable trajectories. While universal for all kinds of simulations, this representation is oftentimes difficult to understand. In the field of multi-body simulations, 3D visualizations present a way of displaying vast amounts of numerical data in an intuitive way which is instantly understandable, even by people without specialized knowledge. For integration of visualization in Modelica multi-body simulations, the "DLR Visualization Library" has been developed. This paper presents the newest additions to the library and shows their application in several DLR projects.

Keywords: Visualization, Simulation, DLR Visualization Library, Modelica, interactive Simulation

1 Introduction

The visualization of simulation data is an important element of advanced simulations. With the increasing complexity of modern multi-body simulations, concerning for example flexible bodies, thermal dissipation, or contacts, the demands for a realistic, real-time capable visualization of the simulation also rise. Using Modelica as modeling language allows the user to pack model functionality in reusable sub-models, e.g. as a replaceable block for a suspension, an engine, etc. This also enables the integration of visualization definitions into the single sub-models, eliminating the need for an additional visualization definition in a separate program.

In 2009, the "DLR Visualization Library" for Modelica has been introduced under the now deprecated name "DLR External Devices Library" [1]. The library features a variety of visualization blocks to be used directly as visualizers within Modelica. Visualization elements like configurable rigid bodies (e.g. sphere, box, gearwheel) or rigid and flexible bodies generated from CAD files can be displayed by connecting their respective Visualizer block to the corresponding frame in the multi-body model. Figure 1 shows an example scene created in Modelica and its visualization. Using the C-interface of Modelica, the visualization information is transmitted to the external visualization software which is started automatically

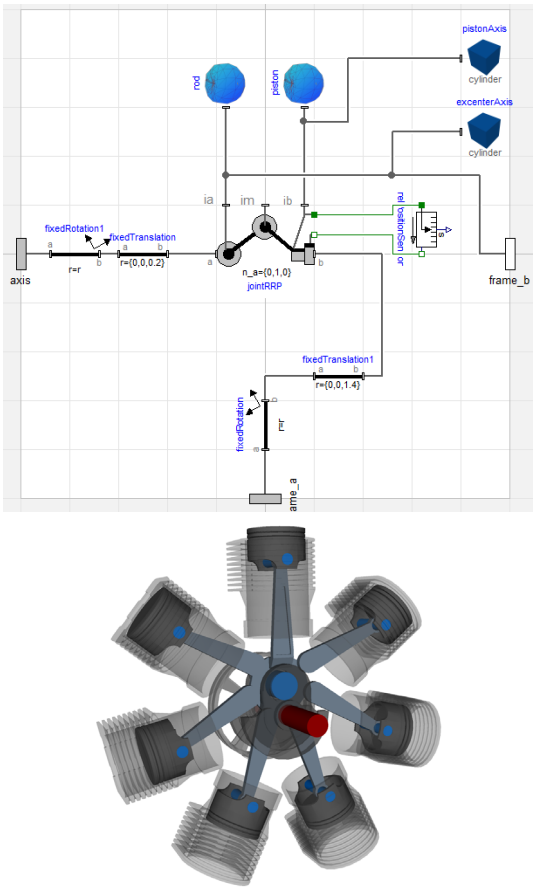

Figure 1: Top: Modelica model for a single cylinder, showing the integration of model and visualization. Bottom: The corresponding 3D visualization of a radial engine.

with the simulation and allows the user to replay and observe the simulation run in real-time and export it as video. Furthermore, as a result of the latest development, the library features a feedback channel, allowing the visualization to influence the simulation. This makes it possible to interact with the simulation using a graphical user interface (GUIs / HUDs), defined in the Modelica model, and to evaluate collisions between visualization elements, this is for example required in contact force calculations.

The following sections shall introduce the newly added features to the "DLR Visualization Library" and present some application examples of its use in ongoing research topics at DLR.

2 State of the art

The first attempts to create 3D Visualizations with Modelica was in 2000, when Engelson [2] started a discussion about techniques for the integration of visualization information with Modelica code. In his paper he presents a set of annotations for 3D graphic primitives and standardized simple geometries. Visualization would then have to be implemented by IDE tool vendors. Even though it brought up the idea of creating visualizations from Modelica models, the presented techniques were never implemented on a larger scale.

Yet, the idea of creating 3D visualizations from Modelica simulations persisted and in 2003 Otter et al. [3] revived the "MultiBody Library" of the Modelica standard library. This change introduced a new sub library called "Visualizers", containing shape objects for simple 3D forms as well as more complex objects from files. These are Modelica blocks and as such may easily be integrated in existing models or may be used in the creation of submodels which contain both simulation and visualization information. The visualization itself is redirected to the ModelicaServices library, responsible for all vendor specific implementations. So the Modelica standard library provides a standardized visualization description, to be implemented by Modelica IDE vendors. This technique is now part of all major simulation environments.

In 2008 Hoeft et al. [4] revisited the ideas of Engelson, this time integrating the powerful X3D standard in Modelica annotations. X3D, short for Extensible 3D Graphics, is an open international standard, developed for web applications. It is a representation of a 3D environment with XML. They present new annotations with X3D code and show an implementation of the technique in the MOSILAB simulator.

Furthermore the Modelica3D library was presented by Höger et al. at the 2012 Modelica Conference [5]. It implements a scene graph in Modelica and uses the C-interface to connect the simulation front-end with the visualization back-end via interprocess communication. Besides their Modelica library, two different back-ends are presented. One based on the OpenSceneGraph 3D library and one based on the Blender 3D graphics software.

Unsatisfied with the existing visualizers, provided by "MultiBody Library", we implemented the "DLR Visualization Library". It is based on the ideas of Engelson and Otter, but tries to take those to the next level with more complex 3D environments, visualizations of mass and power flow, flexible deformation of objects and many other features in a high fidelity rendering.

3 New Functions

3.1 3D-Elements

3.1.1 Dynamic textures

Given a 2D Matrix of 3D points, the flexible surface element may be used to display arbitrary shapes, with the ability to deform during the simulation. More details may be found in [1].

A new addition to flexible surfaces is the ability to display videos as textures, both local files as well as video streams from a network, as shown in figure 2. For local files the video is synced with the simulation time stamp. For network streams this is not possible and the surface will always display the last received image. The supported network protocols and the URL definitions for opening streams are inherited from the underlying FFmpeg library and are described in [18] in detail. Most commonly used protocols are thereby supported, such as: RTP, FTP, MMS, HTTP and HLS.

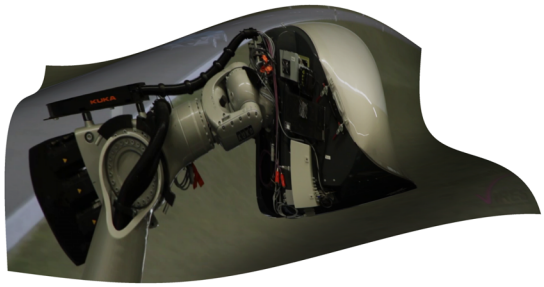

Figure 2: Full HD MPEG Video image from a presentation of the DLR Robotic Motion Simulator [9], rendered onto a flexible surface.

DOI 10.3384/ECP14096899

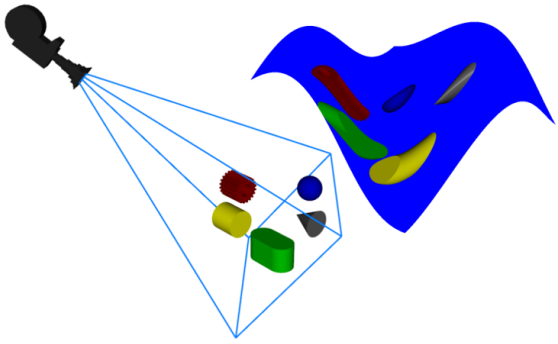

Figure 3: On the left a simple 3D scene; on the right this part is rendered on a flexible surface using a virtual camera, symbolized on the upper left side. All parts are in the same scene.

Besides Videos, images from virtual cameras can be rendered onto a flexible surface. For this purpose a standard camera from the "DLR Visualization Library" is placed in the scene and its output mode is set to "render to texture". The camera output can then be selected as texture for the flexible surface. The result is a camera image, drawn on the surface in the same way it would otherwise be drawn on screen. An example of this is illustrated in figure 3. The whole image shows one scene. On the left side a few objects are arranged and on the right side a flexible surface is depicted which shows the left part of the scene again, as seen from the virtual camera above the arrangement, rendered as a texture on a flexible surface.

3.1.2 Feedback - HUD

In [1] all communication was strictly from the simulation to the visualization. The following, new elements abandon this concept. They not only send data from the simulation to the visualization but also receive data, which may then be used to influence the simulation.

The first interactive objects presented here are interactive HUD elements. The base class for all of these elements is the button class. This defines an invisible HUD object in the visualization and with outputs in the simulation which are dependent on user input on the visualization. By combining the button object with HUD elements for the representation and Modelica logic for reactions, typical user interfaces, like buttons or sliders, can be created and used as interactive input for simulations. The button base class is capable of reacting to mouse-over events while the mouse cursor hovers over them, mouse clicks and dragging of the element by moving the mouse while a button

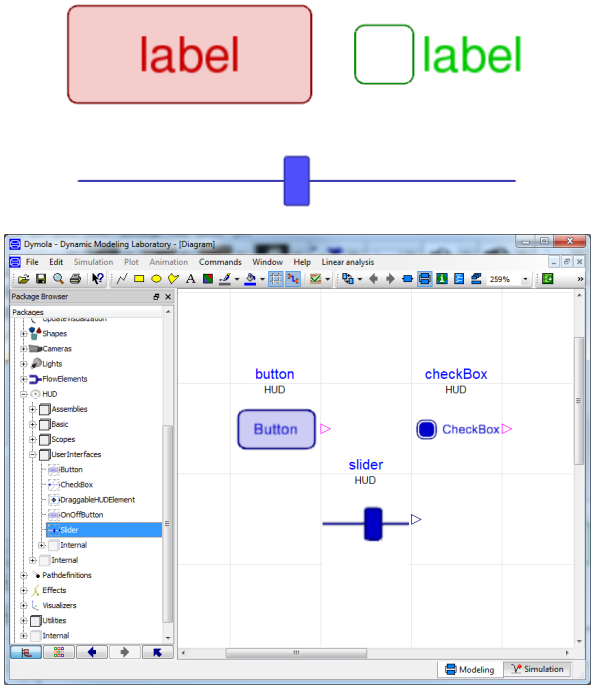

Figure 4: At the top part simple HUD elements are presented and below that their representation in the Modelica graphical designer.

is clicked. The "DLR Visualization Library" contains with a selection of predefined GUI elements, using the described button base class. The following HUD elements are depicted in figure 4:

- Buttons are by default visualized as simple squares with a label. They have a Boolean output value which is "true" for as long as the user presses it and "false" otherwise.
- Check boxes have a different look but actually behave very similar to buttons, except the output value behaves like a flip-flop. Permanently changing its value with every click.
- Sliders for simple adjustment of continuous values

3.1.3 Feedback - Collision detection

The description of complex 3D geometries usually uses geometric meshes. Reading the according data from files, interpreting it and running collision detection algorithms from Modelica is complicated. Physical simulations, especially in the area of multi body simulations often require to measure the distance between objects or contacts between objects. The required data, especially the interpretation and arrange-

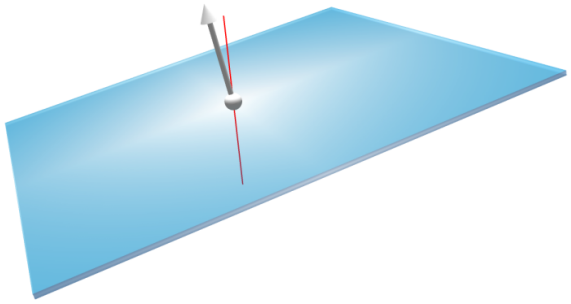

Figure 5: Collision detection with visualization of the contact. The red line represents the collision detector, the sphere shows the contact point and the arrow displays the surface normal.

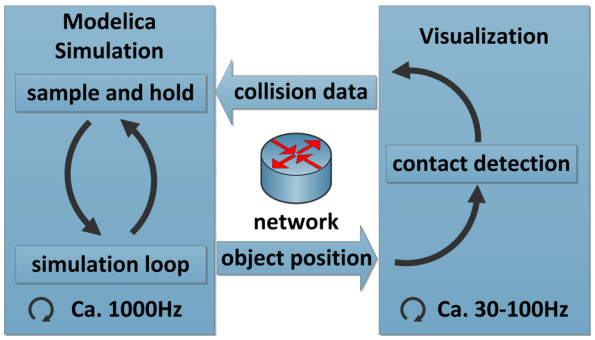

Figure 6: High level view of the collision detection system

ment of 3D mesh data, is already present in the visualization. To make this data accessible for the simulation the collision detector object is introduced. The collision detector is a line in the 3D scene with a defined start and end point. If the line intersects another element in the visualization it produces an output for the simulation. The output data includes a Boolean value, indicating whether or not a collision occurred, the distance of the first collision, measured from the start point and the surface normal vector of the colliding object at the position of the collision. The collision detection may also be visualized as shown in figure 5. The simulation and the visualization are running independently at different frequencies. In our example (see section 4.3), the Modelica simulation of the contact forces runs at a rate of $1kHz$, but the visualization calculating the collisions is bound to the frame rate of the graphics card and is as such dependent on the complexity of the scene and the hardware. By subsampling the communication calls from the simulation to the visualization, a minimum frame rate of about $30-100Hz$ for the visualization is defined. If the visualization fails to achieve this frame rate, the simulation is slowed down below real-time speed as it has to wait for the answer of the visualization system providing the collision data. Because the simulation requires the collision data for each integrator step, a sample and hold interpolation is performed on the collision data, which saves the last received collision data until new ones are available. A high-level overview of this principle is depicted in figure 6.

3.1.4 Weather effects

To enhance the realistic impression of the simulation, the "DLR Visualization Library" provides a selection of the most common weather effects for the simulation. These include: Rain, snow and fog. An example of the rain effect is shown in figure 7. For precipitation both the intensity of the effect as well as the wind direction and speed can be set. Wind can be used to simulate the effects of wind on the direction of precipitation. Fog is parameterized with three values: a starting distance for the fog effect and an end distance, where the view is completely blocked by the fog. Also the color of the fog can be adjusted. The applicability of those effects is not necessarily limited to the simulation of weather but could for example be used in under water simulation to show the effects of murky water.

Figure 7: An exemplary scene with rain effect

3.1.5 Paths

Visualizations often require complex motions of cameras and objects through space. This is type of predefined movement is seldom used in multi body simulations and therefore it is complex to define using the Modelica standard library. To simplify the creation of visualizations, new path definitions are provided. Paths are a multi-body connector frame with a time dependent output. Four different kinds of paths are

DOI 10.3384/ECP14096899

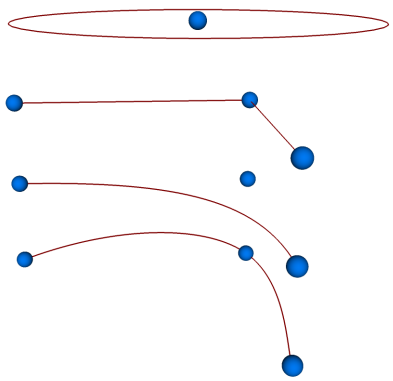

Figure 8: Top to bottom: circular, linear, Bezier, cubic spline; spheres: control points.

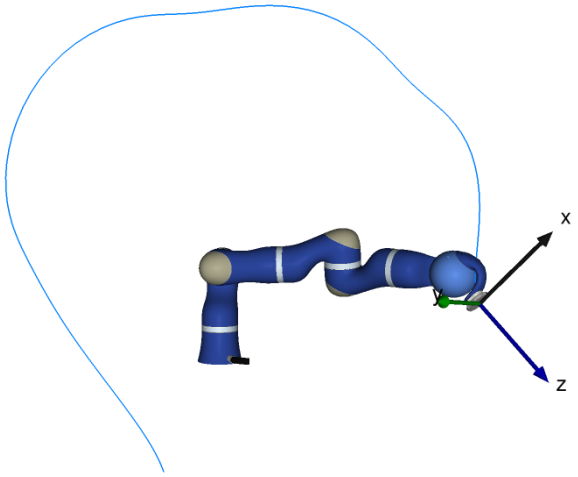

Figure 9: Movement of a robots tool tip visualized using the trace shape.

available as shown in figure 8: The first type shown here is a circular movement. The circle is defined by a center point with orientation and the radius. The simplest paths are point to point connection. A list of points is supplied and the resulting path is a linear interpolation between those. For smoother paths, Bezier curves can be used. The path is only guaranteed to pass through the first and the last point the remaining points only "stretch" the path towards them. The fourth kind of paths is based on cubic spline interpolation. Like Bezier curves these provide very smooth paths. In contrast those the line is always guaranteed to pass through all points. This often simplifies the definition of paths but might lead to overshooting in certain constellations.

3.1.6 Trace shape

Following the 3D movement of multiple objects at once can be hard. To display this motion, a trace-shape can be attached to any object. The trace-shape object then generates a line trail as the object it is attached to moves through space, thereby intuitively representing the objects trajectory. For the illustration of multiple trails, the line thickness, color and length can be adjusted by the user as needed. An example can be seen in figure 9.

3.1.7 Sky-Box

"Empty" space, areas on screen in which no object is defined, are by default filled with a solid color. This is suitable for small animations or abstract illustrations but for many applications a more realistic representation is required. For example in outdoor scenarios those areas should show the horizon and the sky. For this purpose the sky box is introduced. The sky box is always depicted as infinitely far away and is composed of two layers. The front layer is used for displaying some sort of horizon. An example would be distant hills or a tree line. The user has to supply six images (It is designed as a cube with the view point in the middle. One image per side). Behind this layer, visible through transparent areas in the images, the sky is drawn. The user has to provide a date, the time and a position on earth in form of longitude and latitude. The OsgEphemeris library [13] then uses this information to calculate the correct position of the sun and the moon during the day and an astronomically correct star field during the night time. The result is a realistic background for the simulation.

3.1.8 Particle System

The particle system is used for displaying objects, consisting of many sub-objects and cannot be modeled as traditional meshes. Examples for these kinds of objects include streams of water, fire, smoke and dispersed dust. Those objects are visualized using a large number of small objects, showing a simple image, called particles. Those particles are send out randomly from an emitter, follow a certain path while potentially turning, changing color and transparency, until the path reaches a certain length and the particle disappears. Using the right image, a large number of objects, and a specific movement, this raises for example the impression of water flowing from a pipe.

The Modelica blocks for modeling particles are divided by emitter type. The emitter is the area, from which the particles are shot. The three types available in the "DLR Visualization Library" are: Point emitter,

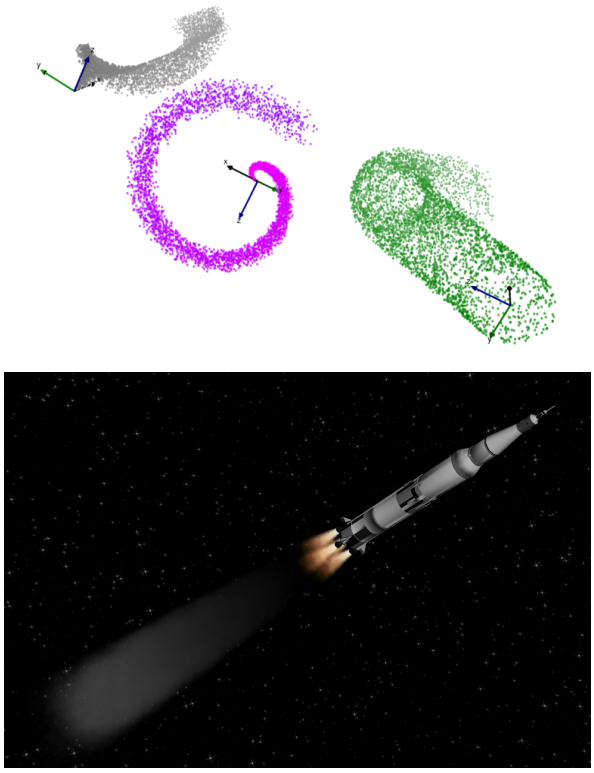

Figure 10: Top: Three types of particle emitters. From left to right line emitter (along z-axis), point emitter, and area emitter (circular shape in $x-z$-plane);
Bottom: Particle effects used to display exhaust from a space rocket.

line emitter and area emitter. Examples for each type are depicted in figure 10. The particles are created randomly within certain bounds, such as mean number of particles per second, configurable for each emitter. Furthermore the particle may change its color, transparency, size over its life time. The final set of parameters then determines the path particles will follow.

3.1.9 Virtual Planet Builder and large scale terrain visualization

The visualization engine is based on the open source library OpenSceneGraph [12], allowing its use in conjunction with another OpenSceneGraph application, which is itself not part of the library: Virtual Planet Builder [14]. Using digital elevation data, like it is commonly produced in aerial surveys with special cameras, Virtual Planet Builder can produce 3D sceneries at scale of whole planets. This is accomplished by auto generating different levels of detail and by converting the scenery in a special file format. The planet surface is tiled into junks. When viewed from a faraway distance large areas are combined into one large tile with very little detail. As the camera moves closer to a certain area the corresponding tile is split into smaller junks with more detail. This process is reiterated while the camera closes in until a maximum degree of detail is reached. For the viewer the switchover is not visible for the lower level detail at which a tile is rendered from afar, is not visible. Of special importance is the way the data is preprocessed and stored in a special format, allowing the renderer to load certain parts at different levels of detail as needed, very efficiently. This technique makes it possible to show extremely large areas while retaining a high level of detail. Figure 11 shows an earth model, created with Virtual Planet Builder from satellite images.

When rendering these planetary scale images in conjunction with small, close up objects, in setups like the satellite simulation in figure 11, graphical glitches appear. During the rendering process, each pixel is assigned a depth value to determine how objects overlap each other from the cameras perspective. The technical implementation of this uses a so called depth buffer or z-Buffer which safes each pixel distance from the camera (the depth or z-value). This is a, typically 24-bit on modern machines, fixed point value in the range $[0,1]$, where 0 represents the near plane (minimum distance from camera) and 1 the far plane (maximum distance) of the view frustum. Therefore, with increasing distance between the near and far plane, the depth buffer resolution decreases. When the depth value of two points is so small that the depth buffer is unable to represent the difference graphical glitches become visible [15]. The minimum depth difference representable by the depth buffer Δz_{min} can be calculated with equation (1). z is the depth value, n is the near

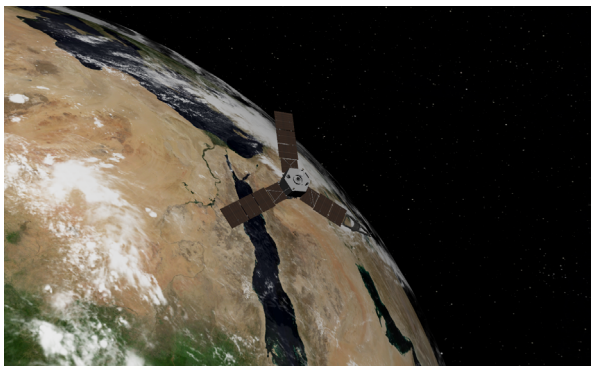

Figure 11: A model of planet earth, generated using virtual planet builder, in a satellite simulation; Logarithmic Z-Buffering allows for a detailed satellite model in the foreground and an earth model at real scale behind it.

DOI 10.3384/ECP14096899

plane distance and b is the depth buffers resolution in bit. The higher resolution at low distances is caused by perspective effects.

$$\Delta z_{min} \approx \frac{z^2}{2^b n - z} \tag{1}$$

Logarithmic depth buffers are a well known technique which seeks to attenuate this problem by applying a logarithm to the depth value before it is written to the depth buffer, thereby increasing the resolution for close objects, where small difference are more likely to be important at the expense of the resolution for distant objects where small differences are less likely to become visible. Equation (2) show the conversion from the original value z to the new value z_{log}. The addition parameter C is used to adjust whether close or distant objects are preferred.

$$z_{log} = \frac{2\ln(Cz+1)}{Cf+1} - 1 \tag{2}$$

Using a logarithmic depth buffer alters the depth buffer resolution. It can now be calculated with equation (3). [16]

$$\Delta z_{log_{min}} \approx \frac{\ln(Cf+1)}{(2^b-1)\frac{C}{Cz+1}} \tag{3}$$

For example, the satellite visualization requires a render distance of $1m$ to $15000km$. With a $24bit$ depth buffer this results in a minimum depth separation of less than $1mm$ at the satellites distance of $100m$, but between $\approx 550m$ and $\approx 1100km$ at the earths distance of $300km - 13000km$, causing visible problems with its rendering. Using logarithmic depth buffers with a C Value of 0.001 the minimum separation for the satellite also lies below $1mm$, yet for earth it is in the range of about $\approx 20cm - 7.5m$ which is well below the visible range at this distance.

3.1.10 Oculus Rift Integration

The Oculus Rift is head-mounted virtual reality display, currently under development by Oculus VR. At the time of this paper, it is only available as a developer preview version with a finalized consumer product in development. The head-mounted system depicted in figure 12 includes a display with a resolution of 1280×800, two fish eye lenses stretching the image to a field of view of about $90° - 110°$ and a three degree of freedom rotational acceleration sensor [17]. With this device it is possible to generate a fully immersive experience in a 3D environment, nearly filling the user's entire field of view and following his motion as he moves his head. The "DLR Visualization Library" supports the currently available Oculus Rift Development Kit as alternative display device for any kind of already integrated cameras. All setup steps and the camera orientation change as reaction to the users head movement are handled fully automatically.

Figure 12: The Oculus Rift head-mounted virtual reality device.

3.1.11 HUDs

Head-Up-Displays are a two dimensional layer in front of the three dimensional scene. The possible applications for this kind of display range from overlaying logos, over the display of model state variables to complex interactive user interfaces. All user interfaces are composed of five base elements: The first element is text. Text can change dynamically and therefore be used to display model states or variables. Also typical text formatting methods such as different fonts, bold text and so on are supported. The second type of elements is a line drawn along a list of user pro-

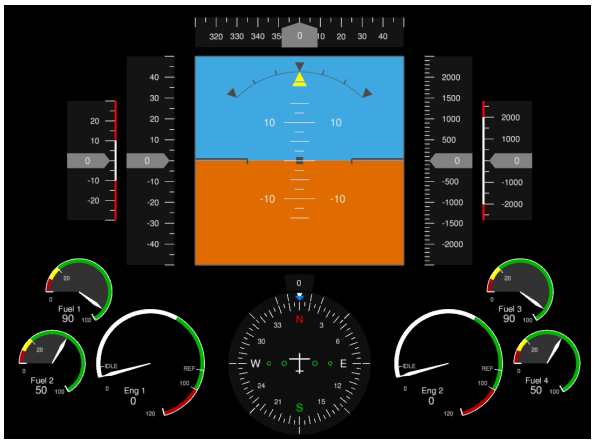

Figure 13: Various HUD elements used to recreate a plane's cockpit instrumentation with artificial horizon, speed and altitude meters alongside a compass and engine status displays.

vided points. Next are free form faces, displaying a filled form based on polygon points provided by the user. Lastly, for more complex images, graphics files can be included. Of course the HUD elements presented here can be combined in the creation of new more complex elements. A small selection of such elements is included with "DLR Visualization Library". An example is a scope for the presentation of variable trajectories, a combination of lines, images and text, or the airplane cockpit instrumentation in figure 13. The last type is an interactive button base class which will be discussed in a later section.

Attention has to be paid to the arrangement of HUD elements for the size of the window they are drawn is variable even at runtime. Therefore it would be impractical to use absolute coordinates and instead relative coordinates are employed. For the adaptation to a changed display size the user can choose of the following four techniques:

1. The horizontal coordinate value spans over the horizontal display size and the vertical value is adjusted to its size is such a way that aspect ratio is preserved.

2. The same as 1 but for the vertical instead of the horizontal coordinate value.

3. The program automatically changes between technique 1 and 2, depending on the smaller side.

4. Relative coordinates range for both horizontal and vertical coordinates range over the whole display size. The aspect ratio might be distorted depending on the display window setup.

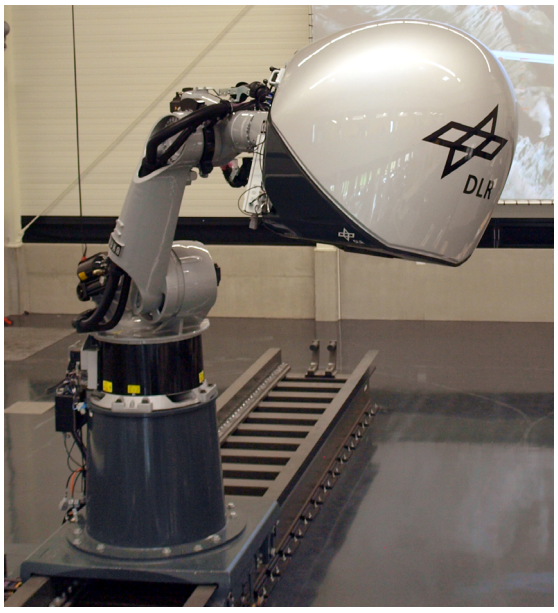

Figure 14: The "DLR Robotic Motion Simulator"

4 Applications

This section exemplifies the real world application of the previously introduced visualization objects with help of their implementation in the "DLR Robotic Motion Simulator". An experimental motion simulation platform, currently under development at DLR. The system is depicted in figure 14.

4.1 ROboMObil GUI

The ROboMObil is a robotic car, developed at the DLR with four separately rotatable wheels, granting a new level of maneuverability. To make use of it, the specialized user interface in figure 15 was developed, using the previously described HUD elements. It is displayed on a touch screen device in the cockpit of the real prototype car

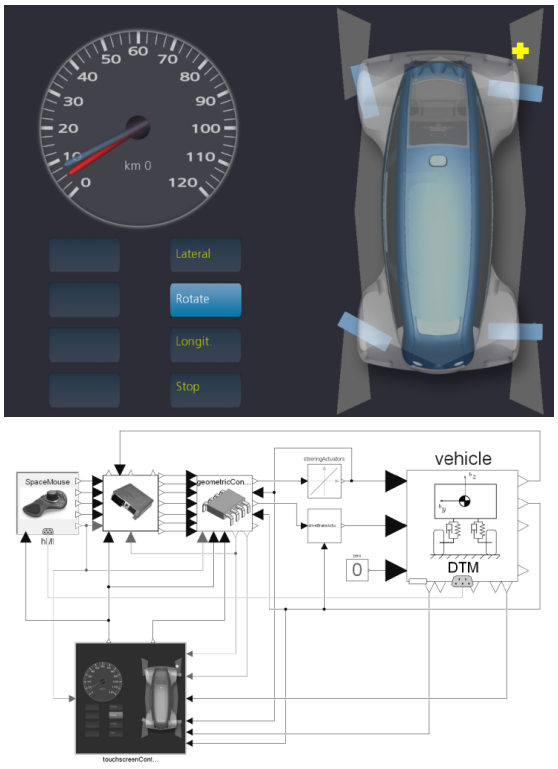

Figure 15: The ROboMObil GUI, as shown on the touch sensitive device of the ROboMObil cockpit, as well as the graphical Modelica designer. Left bottom: the user can choose between different operational modes; right: Enter additional options for the selected mode. In the rotational mode shown here, the user can select an instantaneous center of rotation

DOI 10.3384/ECP14096899

and in the simulators cockpit mockup. The concept is explained at full detail in [8]. The upper left part shows the current and the target speed. Below that the driving mode can be selected and the right part is selecting a rotation center point, required for certain driving modes. The second part of this image shows the full integration of the GUI with other parts of the simulation directly in the model description with Modelica.

4.2 Flexible trajectory planning

The design of a complex trajectory in 3D space, for a predefined movement of objects or cameras, can be a rather difficult and time consuming task, if it has to be accomplished by the direct manipulation of numeric parameters. Therefore the interactive Trajectory Designer has been developed to provide a convenient way of creating control point parameters for a B-spline interpolated movement of position and orientation. The trajectories velocity is interpolated with a two times derivable sinusoidal function.

In order to generate a smooth trajectory through accurately defined bases containing a position and orientation, we use quaternions for orientation and B-Splines for interpolation between these bases. A B-Spline curve $T(x)$ consists of control points $P_i, i \in 1 \ldots n-p$ and B-Spline base functions $N_{i,p,\tau}, i \in 1 \ldots n-p-2$:

$$T(x) = \sum_{i=1}^{n-p} P_i N_{i,p,\tau}(x). \tag{4}$$

The control points P_i each contains seven elements; three for position and four for the orientation represented as quaternion. A base function $N_{i,p,\tau}$ is defined as a polynomial piece with order p and knot vector τ:

$$N_{i,0,\tau}(x) := \begin{cases} 1, & x \in [\tau_i, \tau_{i+1}[\\ 0, & \text{otherwise} \end{cases} \tag{5}$$

$$\begin{aligned} N_{i,p,\tau}(x) \quad = \quad & \frac{x-\tau_i}{\tau_{i+p}-\tau i} N_{i,p-1,\tau}(x) \\ & + \frac{\tau_{i+p+1}-x}{\tau_{i+p+1}-\tau i+1} N_{i+1,p-1,\tau}(x) \quad p > 0 \end{aligned} \tag{6}$$

The knot vector $\tau = [\tau_0, \ldots, \tau_{n-1}]^T, n \geq 2p, \tau_i \leq \tau_{i+1}$ and $\tau_i \leq \tau_{i+p}$ has to be chosen. The algorithms of our implementation set the first and last p knots equal. There are different approaches setting the remaining values, see [11] [10, p161]. For the trajetory designer we use a base functions with degree 3 in order to get a smooth trajetory with a continuity of the second derivative. The parameter x defines the position on the trajectory, $T(x)$ returns the interpolated value $P(x)$ containing the three-dimensional position and an interpolated quaternion. The quaternion interpolation corresponds to a linear quaternion interpolation (LERP)[6]. Therefore the resulting L_2-Norm of the vector is less than 1 and has to be normalized resulting in a unit quaternion which leads to an varying but continuous rotational velocity. Furthermore the trajectory designer provides the functionality to set a velocity at each control point. The behavior of the velocity between the points is computed through sinusoidal functions:

$$v(\lambda) = (\lambda - \frac{1}{2}\pi \sin(2\pi\lambda))(v_{i+1} - v_i) + v_i; \tag{7}$$

with v_i the velocity of the left and v_{i+1} the velocity of the right control point within the interval between P_i and P_{i+1}. The second derivative of this sinusoidal function is continuous and zero at the left and right end providing a smooth transition at the control points. The integration of the velocity v leads to the current position on the trajectory x. Therefore, $v_i \geq 0, i \in 1 \ldots n-p$.

The tool is operated by a small selection of keyboard commands and offers three operating modes to manipulate the position, the orientation and the associated velocity of the control points. In all modes control points can be removed or inserted. For verification of the resulting interpolated movement a live preview, adjusting to the manipulation of control points in realtime, is available. The preview shows a coordinate system traveling along the spline in the main scene and

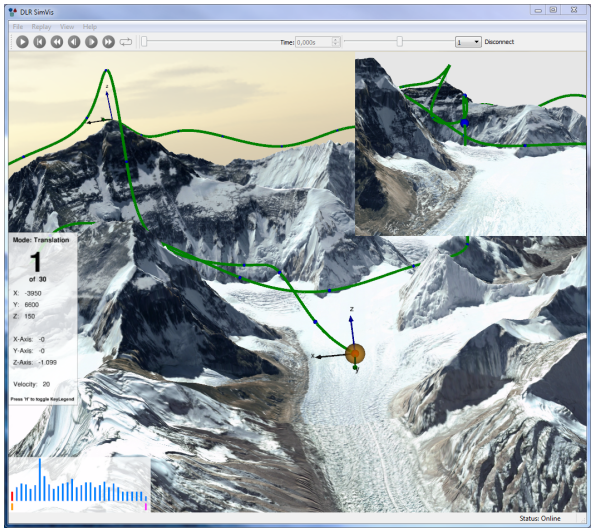

Figure 16: The trajectory designer tool showing a path (green line) above Mt. Everest. In the upper right corner a window previews the camera's trajectory while working on it.

a point of view rendering, adequate for the preview of camera movements, in a separate picture-in-picture preview area. The complete interface is depicted in figure 16. The coordinates, orientations and velocities are finally stored as vectors in a text file, which can be used to reload the control points for further manipulation or later playback.

4.3 Wheel ground contact

The simulation of cars requires the modeling of contacts between wheels and the ground, realized using the collision detector elements. An abstracted model is shown in figure 17. This contact model is then the basis for far more complex models, such as the car shown in figure 18. In this simulation the load on the wheels will shift when driving on uneven terrain and the suspension reacts accordingly. The ground plane in the simulation can be any 3D shape.

The wheel vertical force is calculated according to the following equation. The contact force f is the sum of a spring force s and a damping force d.

$$\vec{f} = \vec{s} + \vec{d} \tag{8}$$

The spring force is calculated using the penetration depth of the collision object and an other object p, as well as a spring constant provided by the user S, to push the wheel away from the other object, along the

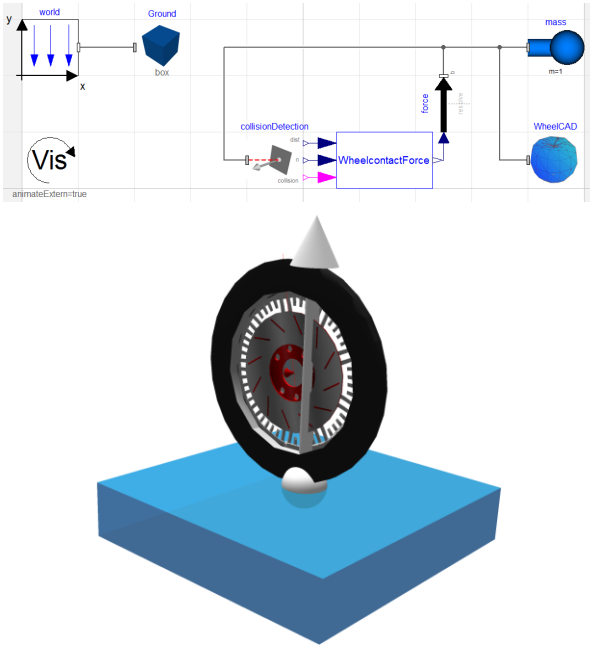

Figure 17: Wheel ground contact in the graphical Modelica designer and as 3D visualization.

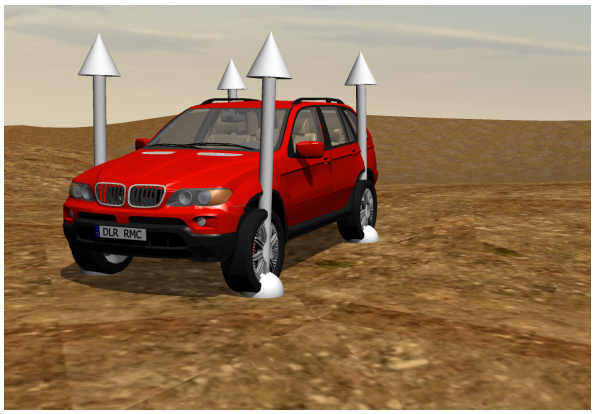

Figure 18: A car driving on a plane. White arrows indicate surface normals for contact points.

objects surface normal at the point of contact.

$$\vec{s} = S \cdot p \cdot \vec{n} \tag{9}$$

Just using a spring force would result in a constantly bouncing wheel. To model energy dissipation, a damping force d is introduced. It is calculated using the collision objects speed $\dot{r}$ in the direction of the surface normal n, a user provided damping constant D and the resulting force, just as the spring force, acts in the direction of the normal. Lastly the damping force should only be present during impact. Otherwise the wheel would act as if it was glued to the surface.

$$\vec{d} = D \cdot \min(0, \vec{\dot{r}} \cdot \vec{n}) \cdot \vec{n} \tag{10}$$

This way of simulating object collisions does come with certain draw backs. First of all, it intrinsically requires the two colliding objects to interpenetrate. While problematic for rigid bodies, it is a reasonable approximation for flexible objects like the car tires in the presented example. When trying to minimize the interpenetration a further problem arises. The larger the spring constant s, the stiffer the simulation gets, requiring ever smaller simulation time steps. Otherwise the interpenetration from one step to the next can be so large, the resulting force from equation (9) gets unrealistically large, hurling the wheel away. The same problem can occur when the wheel is moving with high speed and the ground inclination changes. The sample-and-hold technique used in the communication, delays the point in time when the simulation is able to "see", the changed ground inclination. This is depicted in figure 19. The wheel on the left moves at high speed to the right but due to the slower visualization time steps, the ground inclination change is communicated to the simulation with delay, causing the wheel to penetrate the ground.

DOI 10.3384/ECP14096899

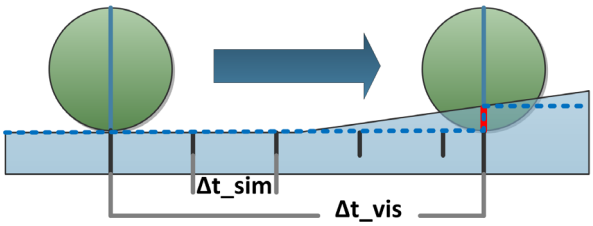

Figure 19: Wheel moving at high speed from left to right. Due to the fact that the simulation does run faster than the visualization, the inclination change is recognized to late. Dotted line: ground level as detected by the simulation; red line: error due to sample and hold technique

For small and fast moving objects, it is even possible for object collisions to be missed completely. The collision detection object only detects collisions with object surfaces so if an object moves so fast towards an other, that the position "jump" between time steps is larger than the collision object length, the collision might get missed. This case is described by equation (11) where v_1 and v_2 the speed of the two objects and i_1 and i_2 are start and end point of the collision object.

$$\left|\left|\frac{v_1 - v_2}{||v_1 - v_2||} \cdot \frac{i_1 - i_2}{||i_1 - i_2||}\right|\right| \frac{1}{f} > ||(i_1 - i_2)|| \qquad (11)$$

4.4 Image warping

The capsule in figure 20 is part of a simulator project. At the back, besides the head of the pilot, are two projectors. The projection screen is the open capsule shell to the top right. The shell has a complex geometry, deforming the images projected onto it. In order to present a rectified image to the pilot a reverse deformation has to be applied to the image prior to projection. This preprocessing utilizes the render image to texture functionality on a flexible surface. Due to the fact that a flexible surface is used, the image can be warped as needed (see figure 20) and with the correct configuration, the final image appears restituted to the pilot.

4.5 Manned vehicle simulation

The "DLR Robotic Motion Simulator" uses the capsule shown in figure 20 and an industrial robot to which the capsule is attached as shown in figure 14. The robot is then used to apply accelerations, in accordance with the simulation, on the pilot, thereby creating an immersive motion simulation. A detailed description of the "DLR Robotic Motion Simulator" can

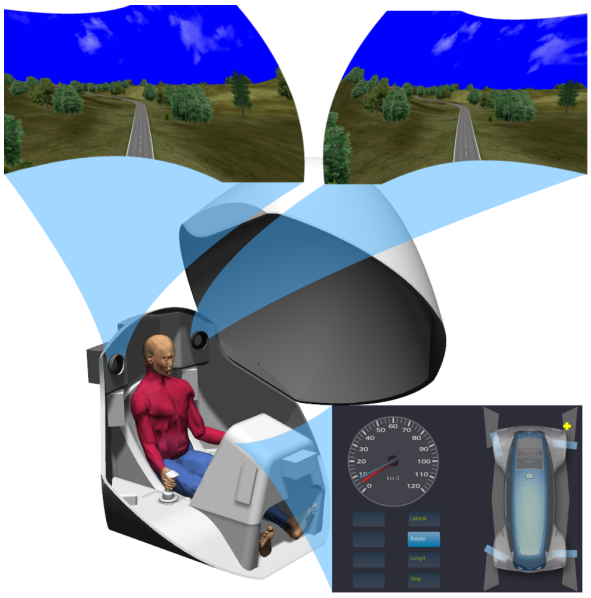

Figure 20: The piloting capsule of the "DLR Robotic Motion Simulator"; On top the stereo images after warping; The images are projected on the capsule and appear restituted; In front of the pilot is a touch sensitive display.

be found in [7]. In this application all of the previously shown applications are utilized. The pilots' main screen is restituted using the render to texture feature on a flexible surface; the console in front of the pilot shows an input GUI on touch screens, and the vehicle simulation uses collision detection objects for the wheel ground contact analysis.

5 Limitations

The presented library does have certain limitations in its current state, of which the following three are currently under investigation for improvements. The first one is the design of the collision detection system: it only allows for collisions with a line object, limiting its use to applications where the point of contact is predictable, such as the presented tire, where the contact point can be assumed to be in the direction of gravity, while arbitrary contact points, like the collision of a car with a pole, can not be modeled in a similar fashion. Also the underpinning architecture, as introduced in [1], only permits retroactive collision detection. It only detects interpenetration of the collision object with an other object after it happened and without any possibility of detecting the exact time of contact. Any contact model relying on the collision data has to account for this. The second item for improve-

ment is the graphical fidelity. While the current system provides very high fidelity compared to other products for scientific visualization, it does not hold up when compared to state of the art graphics as seen in modern computer games. The visualization of simulation data might not require such graphics, yet in virtual reality applications the best graphics possible are desired to maximize the users level of immersion. For comparison between our solution and a modern computer games engine see figure 21. For better visibility a very simple scene was selected here. On the top our solution is shown, with shadows enabled. Below that, the same scene is displayed using the Unity3D engine [19], with deferred lighting, advanced soft shadows, field of view and screen space ambient occlusion among other effects. Even thought it is very simple, the second scene looks more realistic. Finally, the visualization library is based on Modelicas multi-body library. In virtual reality applications a large number of visualization elements gets connected using frame connectors. Even for static and fixed compositions the number of resulting equations gets extraordinarily high. An empty scene, with only the multi-body world object and the visualization libraries updateVisualization object, requires 1073 equations. Each additional ElementaryShape (e.g. a simple box) introduces 217 equations and each fixedRotation object used to arrange the objects in the scene further requires 102 equations. Clearly this modeling is too complicate and for complex simulations it can lead to performance problems. Since described problem is caused by Modelica's design of the multi-body library, we propose a simplification of the connector for the case that no masses are involved, when the multi-body library gets reevaluated in the future.

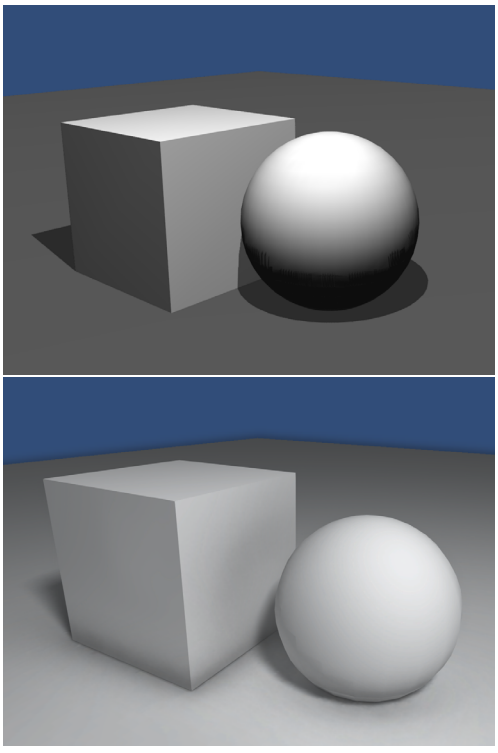

Figure 21: A simple scene for comparison between our current visualization on the top and below a modern computer games engine

6 Conclusion

Visualization is an important, if not necessary, augmentation for a multitude of simulations. The "DLR Visualization Library" provides a sophisticated visualization framework for the Modelica modeling language. The paper presented the new additions to the library: videos and camera images rendered on flexible surfaces, advanced user interfaces, a collision detection system, weather effects, paths, the trace shape, a particle system, sky-boxes and integration with Virtual Planet Builder and support for virtual reality hardware. Furthermore, real-life applications for these new elements were presented as they are used in the development of the "DLR Robotic Motion Simulator".

In comparison with the other existing libraries, our implementation is not based on annotations and therefore does not rely on vendor specific annotations. Also it is not only possible to render all visualizers described in the standard MultiBody library but it also heavily augments its rather limited possibilities. In relation to other solutions, the "DLR Visualization Library" provides the richest feature set along side high fidelity results.

The visualization component is currently developed for Windows XP, 7, 8 and a Linux version is in Beta test with a installation package for Ubuntu 12.04 available. The library it self utilizes the Modelica C-interface and should therefore be compatible with a wide range of simulation environments but currently only Dymola has been tested excessively and is officially supported.

In conclusion, the library proves useful for gaining an intuitive understanding of multi-body simulations, the creation of presentable results and the creation of interactive virtual reality environments.

DOI 10.3384/ECP14096899

References

[1] Bellmann Tobias. Interactive Simulations and advanced Visualization with Modelica. Como, Italy: Proceedings of the 7th International Modelica Conference, Linköping University Electronic Press, 2009.

[2] Engelson Vadim. 3D Graphics and Modelica - an integrated approach. Linköping, Sweden: Linköping Electronic Articles in Computer and Information Science, 2000.

[3] Otter Martin, Elmqvist Hilding and Mattsson Sven Erik. The New Modelica MultiBody Library. Linköping, Sweden: Proceedings of the 3rd International Modelica Conference, Linköping University Electronic Press, 2003.

[4] Hoeft Thomas and Nyscht-Geusen Christoph. Design and validation of an annotation-concept for the representation of 3D-geometries in Modelica. Bielefeld, Germany: Proceedings of the 8th International Modelica Conference, Linköping University Electronic Press, 2008.

[5] Höger Christoph, Mehlhase Alexandra, Nytsch-Geusen Christoph, Isakovic Karsten and Kubiak Rick. Munich, Germany: Proceedings of the 9th International Modelica Conference, Linköping University Electronic Press, 2012.

[6] Erik B. Dam, Martin Koch and Martin Lillholm. Quaternions, interpolation and animation. Datalogisk Institut, Københavns Universitet, 1998.

[7] Bellmann Tobias, Heindl Johann, Hellerer Matthias, Kuchar Richard, Sharma Karan and Hirzinger Gerd. The DLR Robot Motion Simulator Part I : Design and Setup. Shanghai, China: IEEE International Conference on Robotics and Automation, 2011.

[8] Bünte Tilman, Brembeck Jonathan and Ho Lok Man. Human Machine Interface Concept for Interactive Motion Control of a Highly Maneuverable Robotic Vehicle. Baden-Baden, Germany: Speech, 2011 IEEE Intelligent Vehicles Symposium (IV), 2011.

[9] DLR. DLR Robotic Motion Simulator - Driving Simulation.: Website/Video, 2013. *http://www.youtube.com/watch?v=QCGcWOwV8Qs*

[10] Gerald Farin. Curves and Surfaces for CAGD - A Practical Guide. 5. edition. : Morgan Kaufmann, 2001

[11] Gerhard Schillhuber. Geometrische Modellierung oszillationsarmer Trajektorien von Industrierobotern. Munich, Germany: Technische Universität München, 2002

[12] Osfield Robert and others. OpenSceneGraph.: Website, 2013. *http://www.openscenegraph.org/*.

[13] osgephemeris - An ephemeris model for use with OpenSceneGraph.: Website, 2013. *http://code.google.com/p/osgephemeris/*.

[14] Osfield Robert and others. openscenegraph/VirtualPlanetBuilder - GitHub.: Website, 2013. *https://github.com/openscenegraph/VirtualPlanetBuilder*.

[15] Khronos Group. The Depth Buffer. Beaverton, Oregon, USA: Website, 2013 *http://www.opengl.org/archives/resources/faq/technical/depthbuffer.htm*

[16] Patrick Cozzi and Kevin Ring. 3D Engine Design for Virtual Globes. : CRC Press, 2011

[17] Oculus VR, Inc. Oculus Rift - Virtual Reality Headset for 3D Gaming | Oculus VR. Irvine, California, USA: Website, 2013. *http://www.oculusvr.com/*.

[18] FFmpeg team. FFmpeg Documentation.: Website, 2013. *http://ffmpeg.org/ffmpeg-protocols.html*.

[19] Unity Technologies. Unity - Games Engine. San Francisco, California, USA: Website, 2013. *http://unity3d.com/*.