

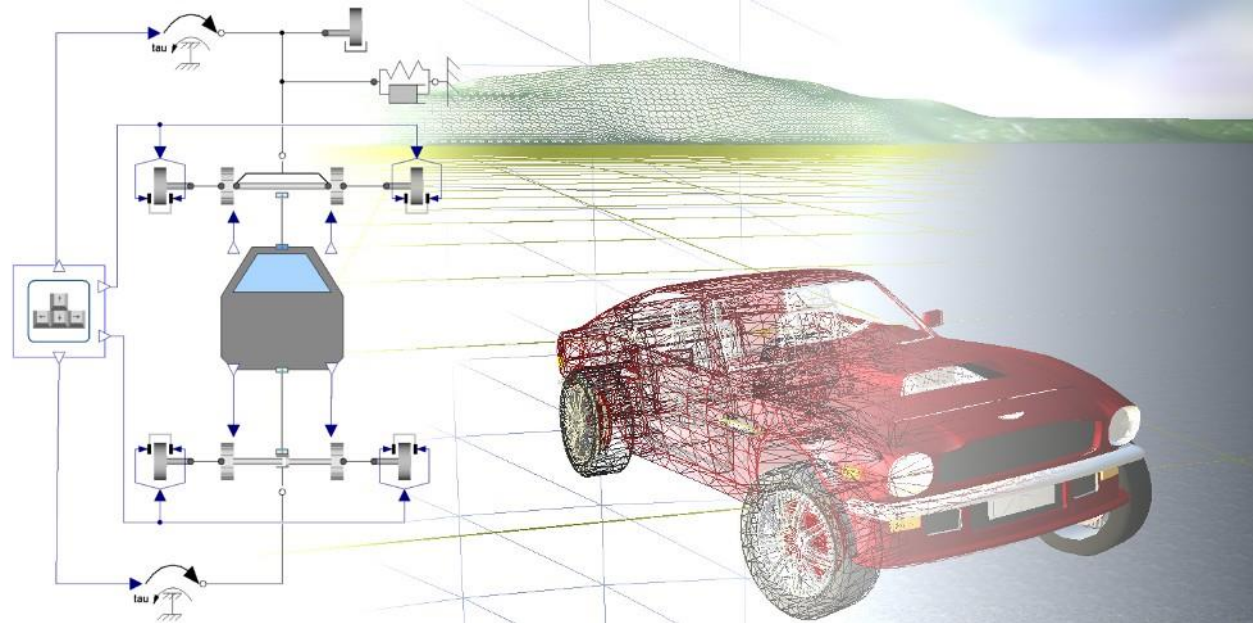
Virtual Physics Equation-Based Modeling

TUM, December 20, 2022

Real-Time Simulation with Dymola

equation

```
sx0 = cos(frame_a.phi)*sx_norm + ...  
sy0 = -sin(frame_a.phi)*sx_norm + ...  
vy = der(frame_a.y);  
w_roll = der(flange_a.phi);  
v_long = vx*sx0 + vy*sy0;  
v_lat = -vx*sy0 + vy*sx0;  
v_slip_lat = v_lat - 0;  
v_slip_long = v_long - R*w_roll;  
  
v_slip = sqrt(v_slip_long^2 + ...  
-f_long*R = flange_a.tau;  
frame_a.t = 0;  
f = N*. S_Func(vAdhesion,vSlide,...  
f_long =f*v_slip_long/v_slip;  
f_lat =f*v_slip_lat/v_slip;  
f_long = frame_a.fx*sx0 + ...  
f_lat = -frame_a.fx*sy0 + ...
```



Dr. Dirk Zimmer

German Aerospace Center (DLR), Robotics and Mechatronics Centre

In this lecture, we give an example of modeling a fully functional real-time simulation. This concerns essentially three topics:

- Time-Integration for Real-Time and synchronization.
- Handling of User Input.
- Real-Time 3D Visualization.

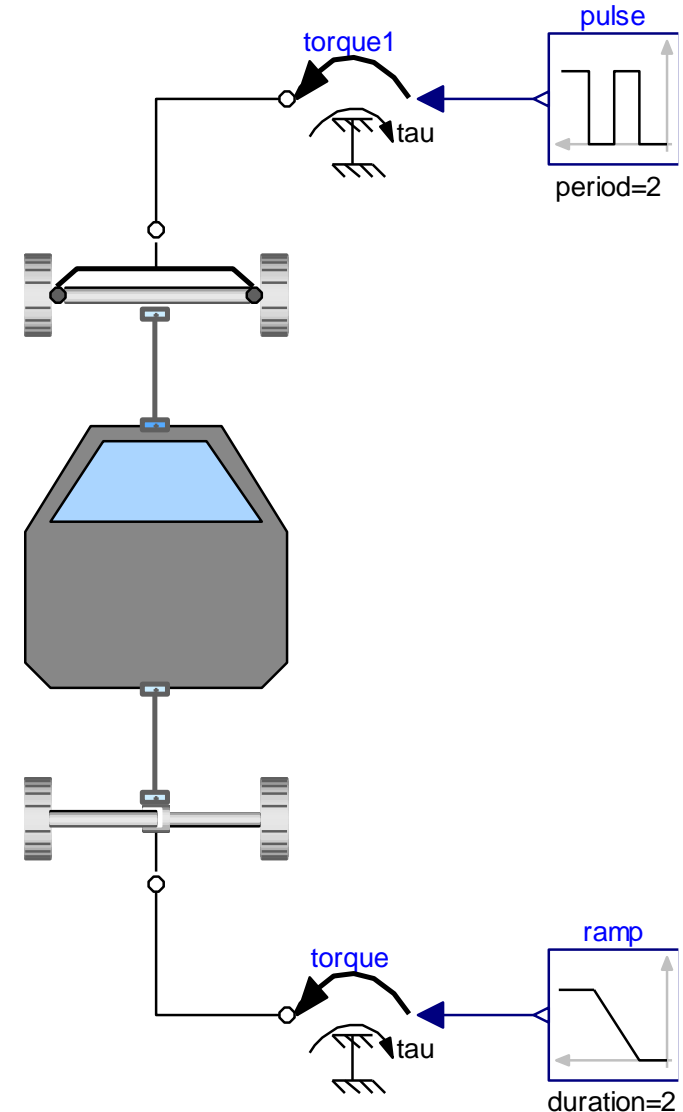
If we want to simulate something in real-time. The numerical ODE-solver is subject to a few severe constraints.

- The solver must compute fast enough
→ larger stepsizes or simple algorithms
- If the system is interactive, there is a maximum step-size
→ favors simple algorithm.
→ fixed step-size methods
- Each single integration step must be fast enough
→ no solvers with indefinite number of iterations (avoid any non-linearities)
→ no events.
→ no implicate solvers (will be explained after Christmas)

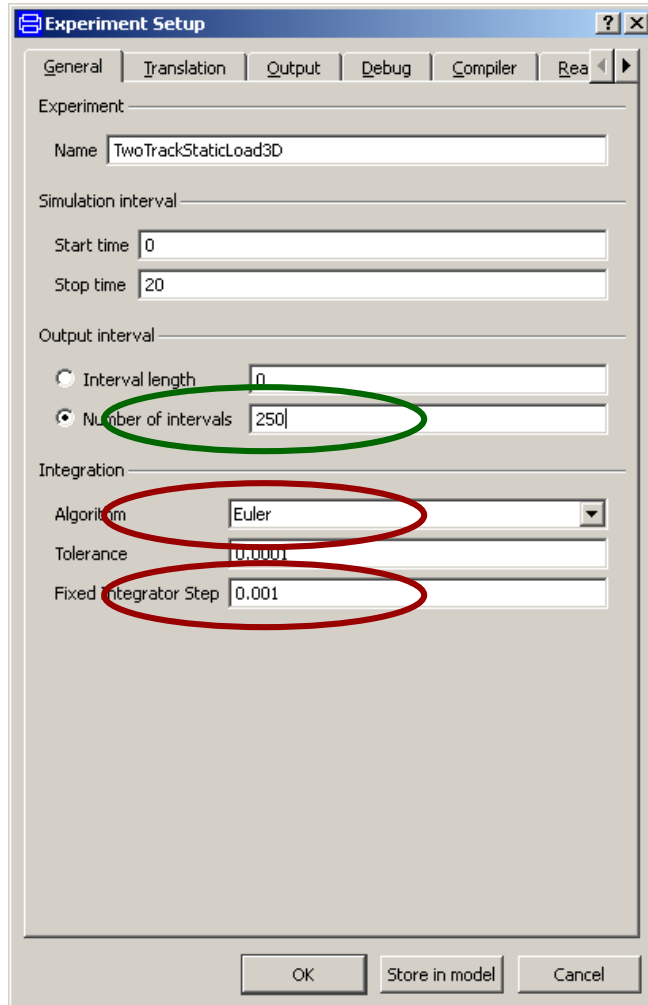
Time Integration

The two-track car model seems to be suited to be simulated in real time.

- Only linear-systems of equations (non-linear solvers are not required)
- No events
- Limited stiffness.



In Dymola, it is very easy to simulate the two-track model in real-time.



In Dymola it is very easy to simulate the two-track model in real-time.

- We simply use the most simple solver that is available:
Forward Euler
- We use a fixed step-size of 1ms
- We may reduce the number of output values (since writing to the disc can easily be more time-consuming than the actual simulation...)
- In fact, we are much faster than real-time. We need to artificially slow-down the simulation in order to synchronize with real-time.

name



Normal

- For real time synchronization, we need a special model.
- This model is contained in the Modelica Device Drivers Library (developed by DLR)
- It slows down the simulation by calling a function that stays in an idle loop.

The Synchronize Realtime

Block: **name**



Normal

- The block simply calls an Modelica function of the DeviceDrivers Library.

```
block SynchronizeRealtime
```

```
parameter Integer resolution(min = 1);
```

```
parameter ProcessPriority p;
```

```
output Real calculationTime;
```

```
output Real availableTime;
```

```
equation
```

```
when (initial()) then
```

```
  setProcessPriority(
```

```
    if (p == "Idle") then -2
```

```
    else if (p == "Below") then -1
```

```
    else if (p == "Normal") then 0
```

```
    else if (p == "High") then 1
```

```
    else if (p == "Realtime") then 2
```

```
    else 0);
```

```
end when;
```

```
(calculationTime, availableTime)
```

```
=
```

```
  realtimeSynchronize(time, resolution);
```

```
end SynchronizeRealtime;
```


The Synchronize Realtime

Block:

name



Normal

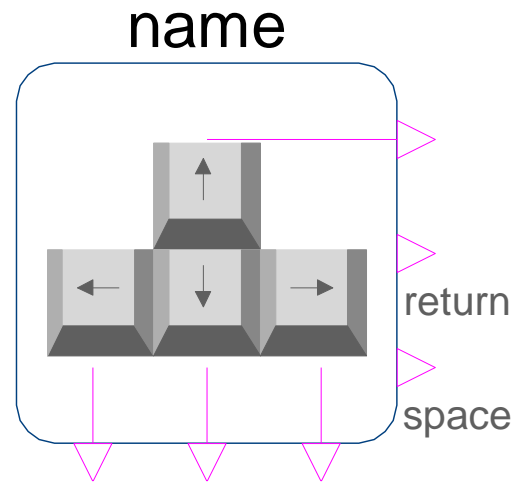
- The block simply calls an Modelica function of the DeviceDrivers Library.

```
function realtimeSynchronize
  input Real simTime;
  input Integer resolution = 1;
  output Real calculationTime;
  output Real availableTime;
  external "C" calculationTime =
OS_realtimeSynchronize(simTime,resolution,
availableTime);

annotation(Include = "
#ifdef MDDSYNC
#define MDDSYNC
#include <windows.h>
[...]

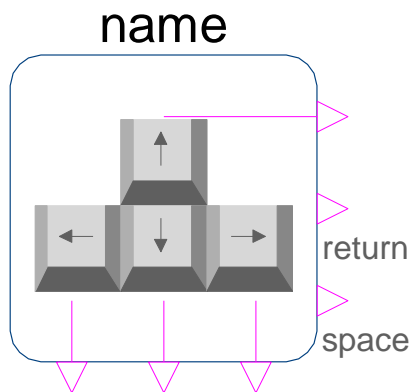
double OS_realtimeSynchronize(double simTime,
  int resolution, double * availableTime) {
  [...]

  while((getTime(resolution)- startTime)/1000 <= simTime)
  {
    Sleep(0);
  }
  [...]
}
#endif
");
end realtimeSynchronize;
```



- Also for the user interaction, we need a special input block.
- This block is contained in the Modelica Device Drivers Library (developed by DLR)
- The Boolean output signals indicate when a certain key is pressed down.

The Keyboard Input Block:



- The block simply calls an Modelica function of the DeviceDrivers Library.
- It simply polls the current state of the keyboard with a given sample rate.

```
block KeyboardInput
  parameter Real sampleT = 0.01

  BooleanOutput keyUp;
  BooleanOutput keyDown;
  BooleanOutput keRight;
  [...]

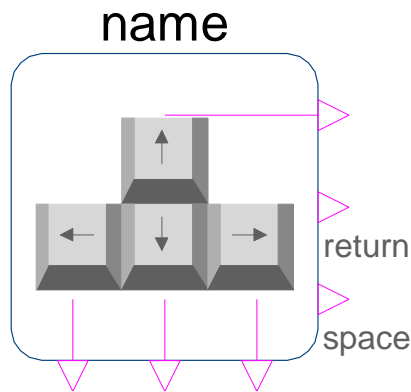
  Integer KeyCode[10];
  InputDevices.Keyboard keyboard;

equation
  when (sample(0, sampleT)) then
    KeyCode = keyboard.getData();
  end when;

  keyUp = (KeyCode[1]==1);
  keyDown = (KeyCode[2]==1);
  keyRight = (KeyCode[3]==1);
  [...]

end Frame;
```

The Keyboard Input Block:



- On the right you see the `getData` function that is called to poll the keyboard state.
- It calls an external C function.
- The code is contained in the annotation.

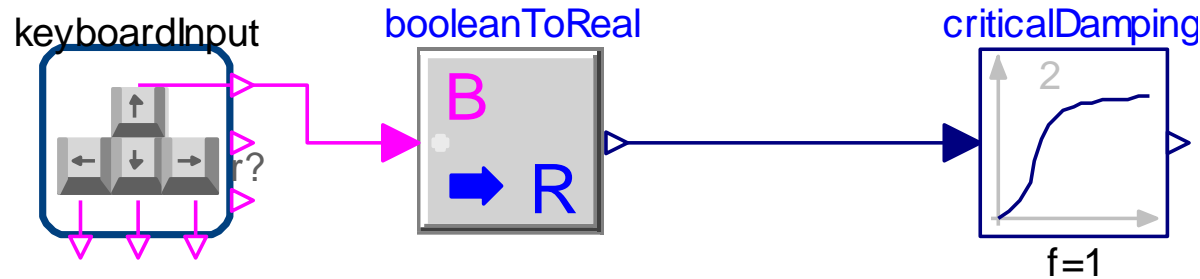
```
function getData

    output Integer KeyCode[10];

    external "C" KEY_getData (KeyCode);

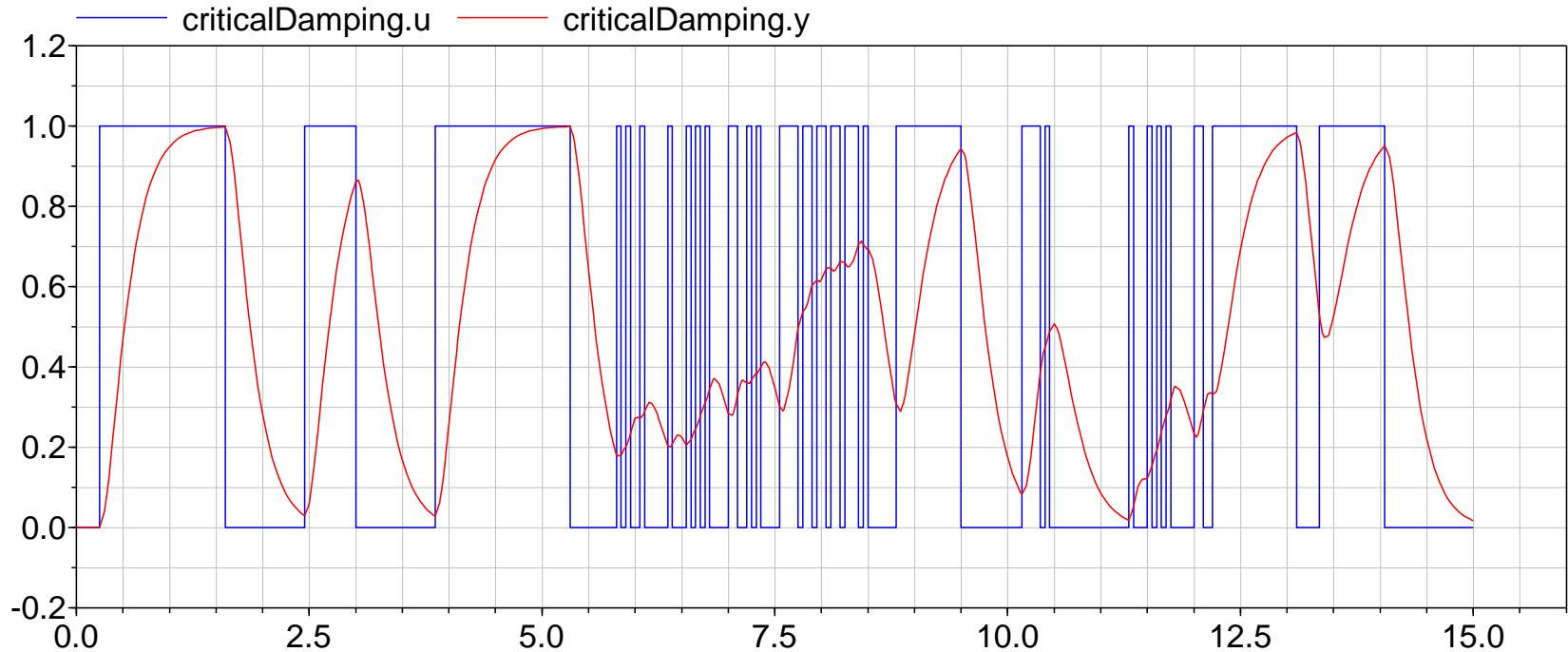
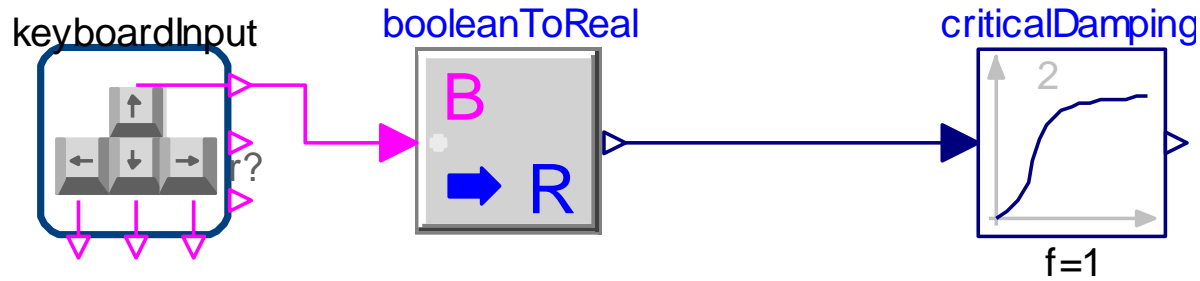
    annotation (Include=""
#define VOID void
typedef char CHAR;
typedef short SHORT;
typedef long LONG;
#include <windows.h>
[...]
void KEY_getData(int * piKeyState)
{
    if(GetAsyncKeyState(VK_UP))
        piKeyState[0] = 1;
    else piKeyState[0] = 0;")
    [...]
    ""

end getData;
```

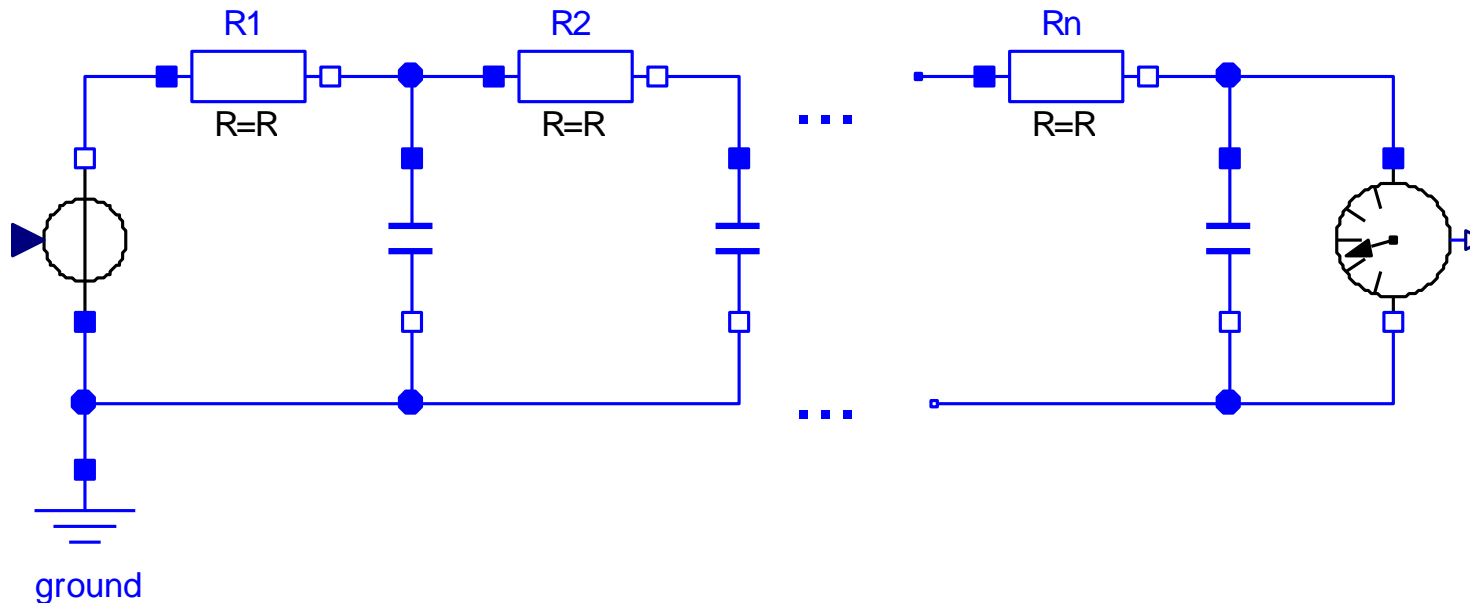


- Using this input block, the user can only control in a Boolean way: ON or OFF.
- To enable a more continuous control, we can filter the input signal.
- To this end, we apply the critical-Damping Filter from the Modelica Standard Library.

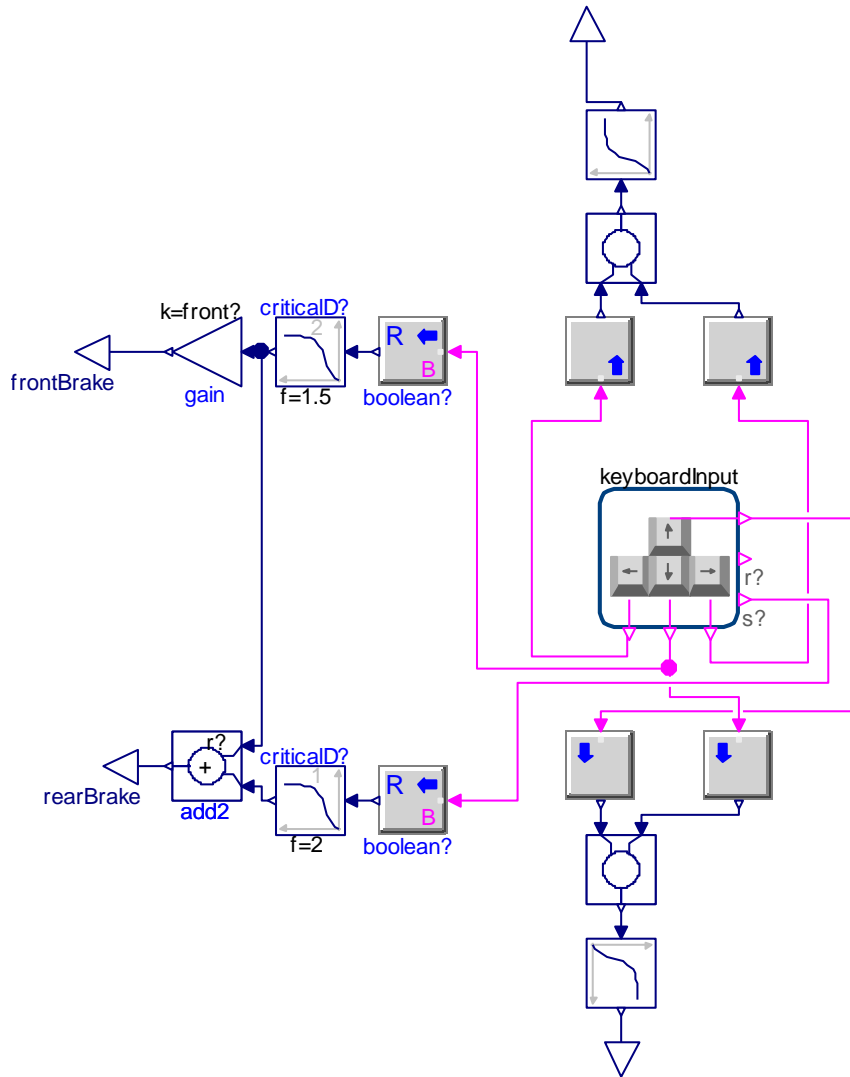
Filtering User Input



n elements

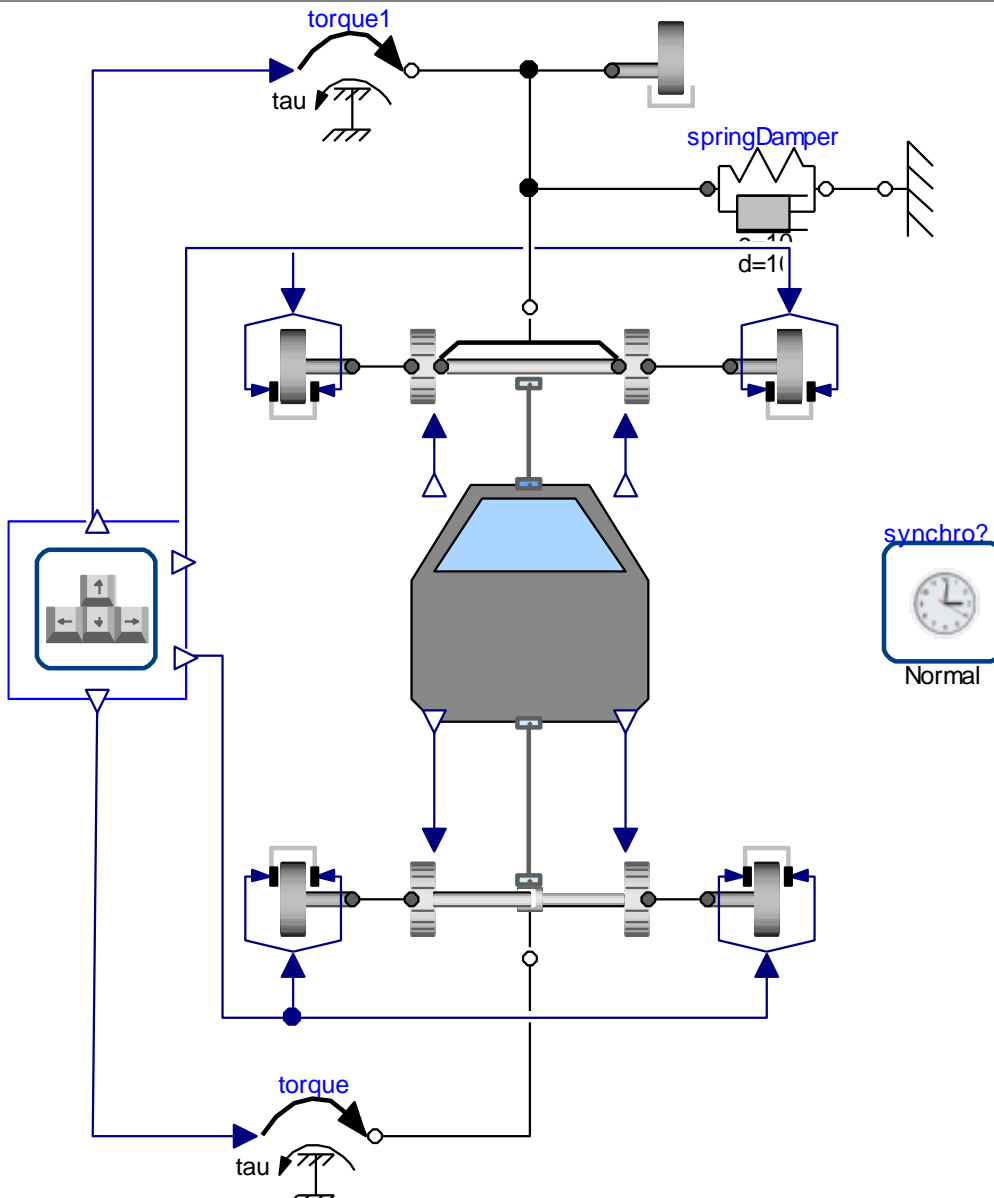


- This electrical circuit illustrates the functionality of the critical-damping filter
- It can be regarded as RC lowpass filter with multiple stages (in our case: 2)



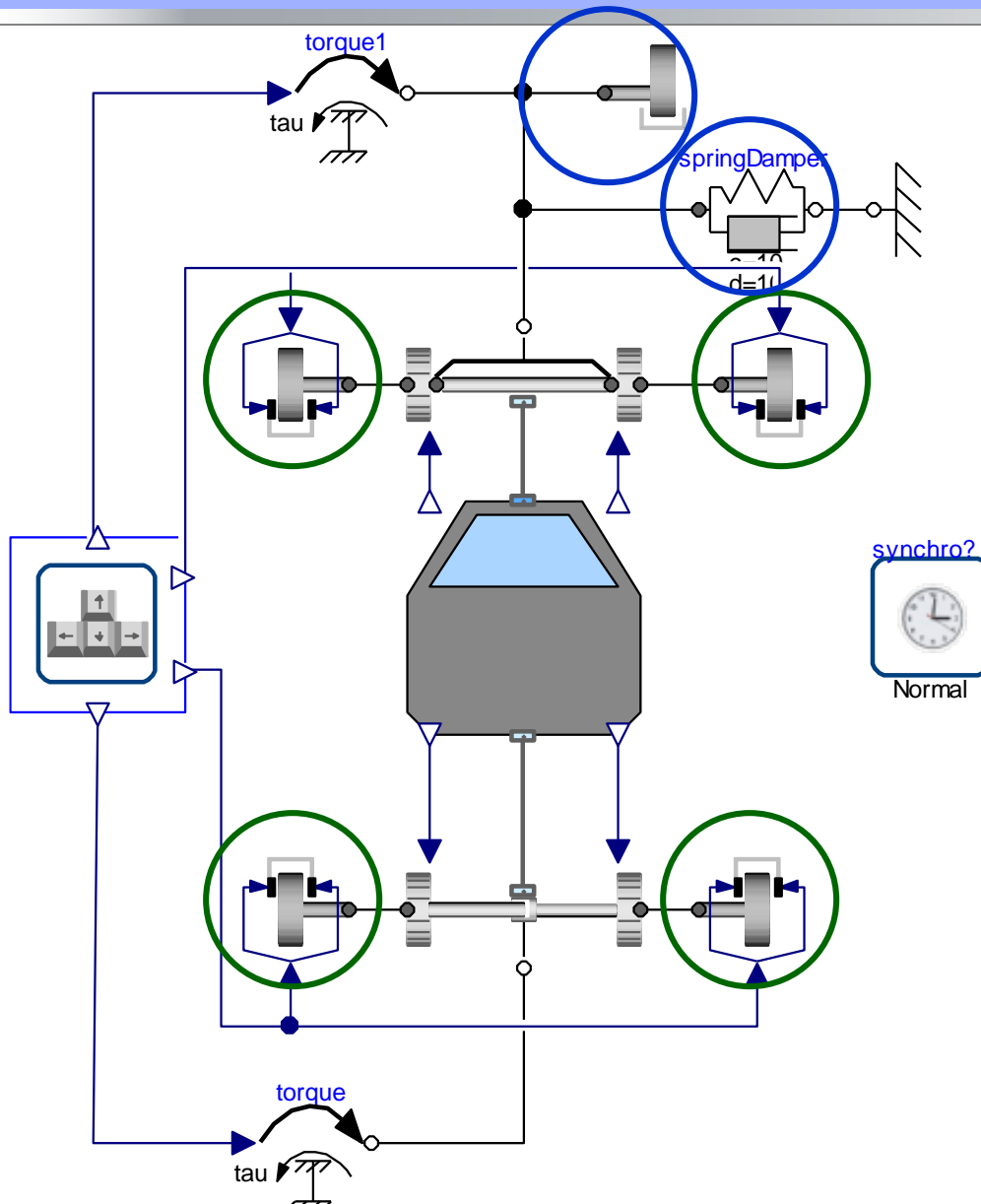
- Using critical damping filters, I created a control block for the car model.
- Its outputs are the breaking forces and the driving and steering torque.

Applying User Interaction



- The forces and torques are then applied on the car model.

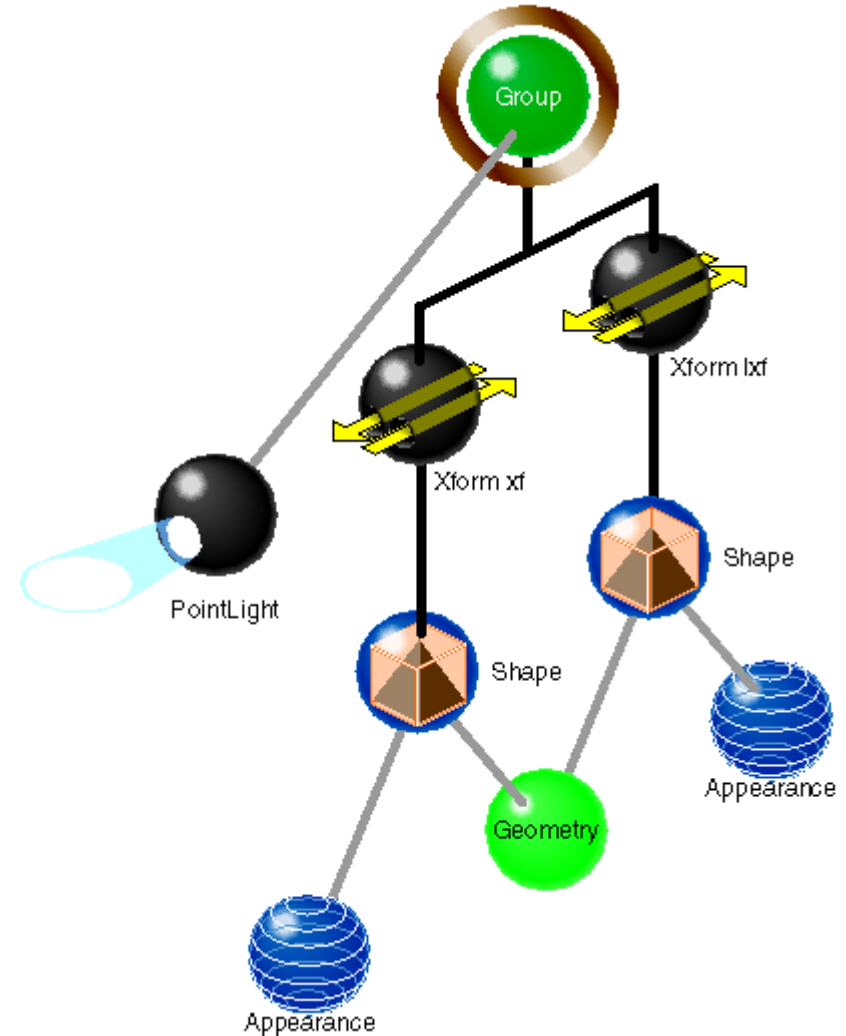
Applying User Interaction



- The forces and torques are then applied on the car model.
- There is simple brake model
- The steering is limited and auto-centered by a spring-damper system.

- Now we can steer and simulate our car model in real-time but this makes hardly any fun, if we do not have a 3D real-time visualization.
- The SimVis Library supports a real-time visualization in 3D. It has been developed by DLR.
- SimVis is based on the OpenSceneGraph Technology that itself uses the OpenGL standard.
- The SimVis library is conceptually similar to the DeviceDrivers library. It provides a set of Modelica models that then call external C-functions.

- OpenSceneGraph is an open source implementation of the scene graph technology.
- In the scene graph technology the scene is describes as a graph.
- The visualization of the graph is based on the OpenGL 2.1 standard.
- For the online-visualization, all we need to do is to update the graph.



The SimVis Library contains various elements:

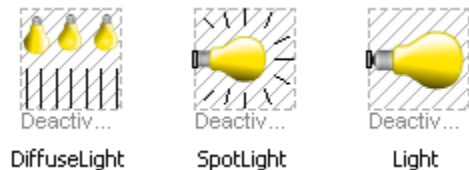
- Shapes



- Cameras



- Lights



The SimVis Library contains various elements:

- Shapes



- Cameras

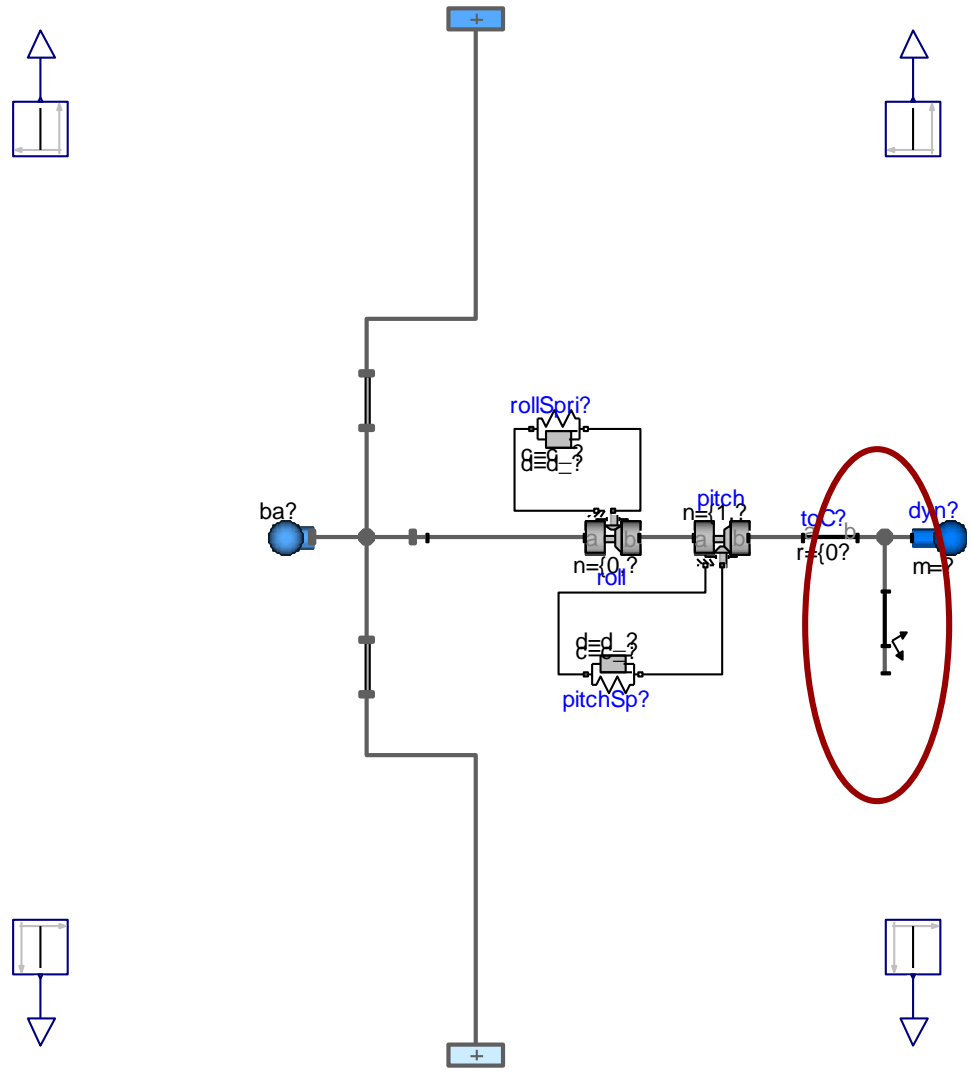


- Lights



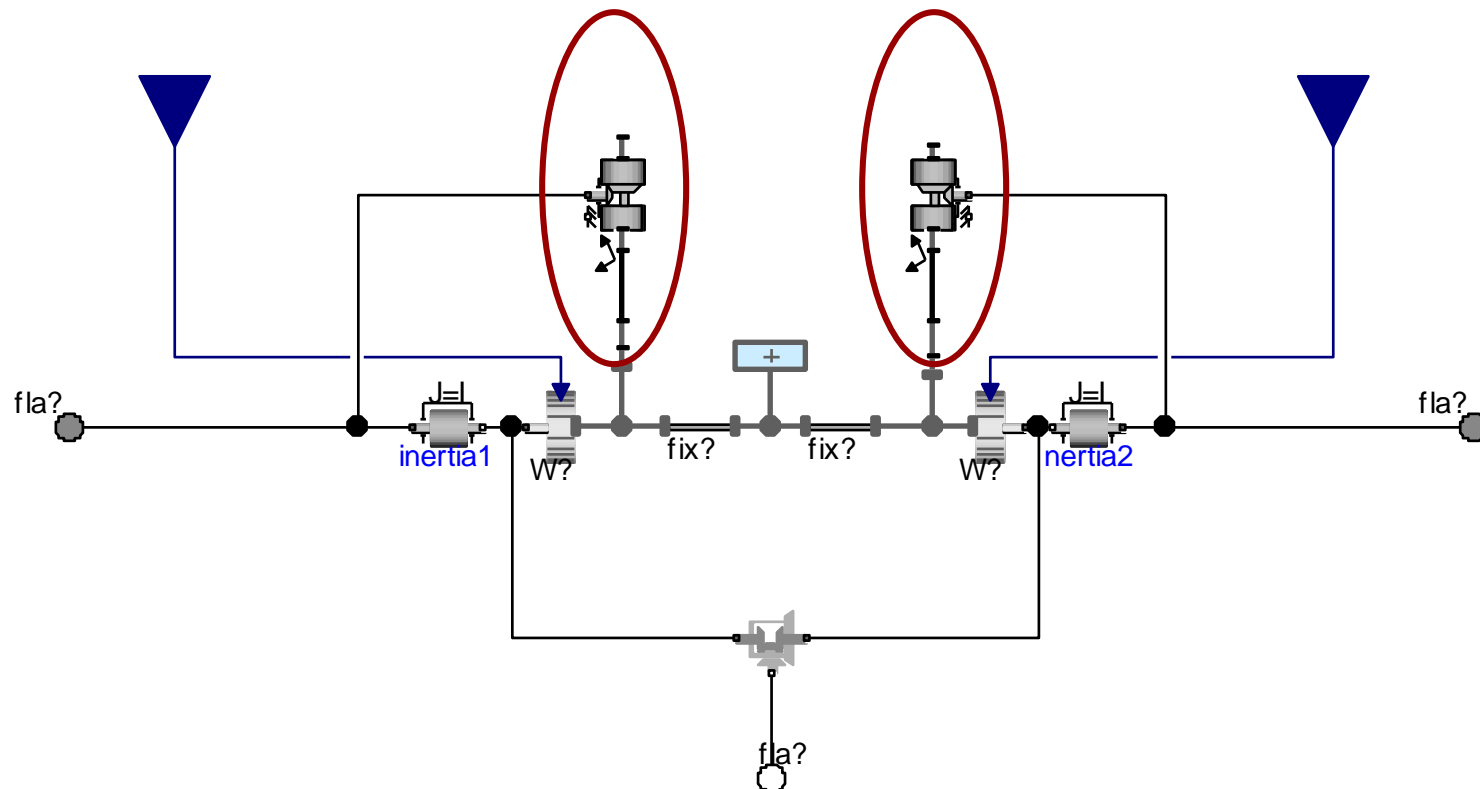
All these elements use the Frame Connector form the MultiBody library.

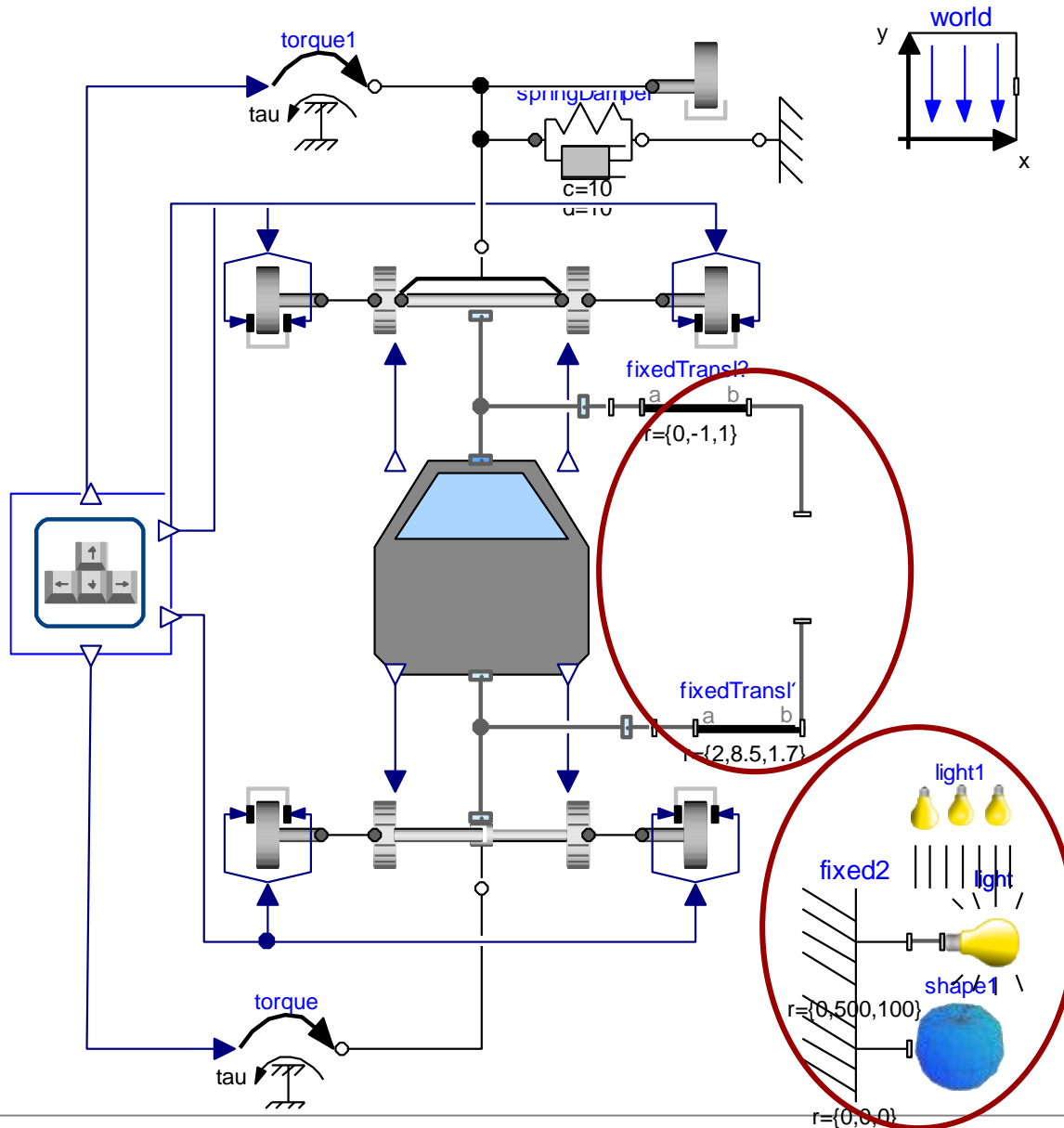
Hence they can simply be used like MultiBody components.



- The visualization of the wheels is integrated into the chassis model

- The visualization of the wheels is integrated into the axis model





- Lights and Landscape are added to form the complete scene.
- A dynamic follow camera is attached to the rear end of the car pointing to the nose.

- And voila!



- We're done! Almost... the rest is your task in Exercise 9.

Questions ?