VISUM: A VR SYSTEM FOR THE INTERACTIVE AND DYNAMICS SIMULATION OF MECHATRONIC SYSTEMS

D. Finkenzeller [1], M. Baas [1], S. Thüring [1], S. Yigit [2], A. Schmitt [1]

(1) : University of Karlsruhe
Institut für Betriebs- und Dialogsysteme
Am Fasanengarten 5
76128 Karlsruhe
Germany
+49 (0)721 608 3965
E-mail: {aschmitt, baas, dfinken, thuering}@ira.uka.de

(2) : University of Karlsruhe
Institut für Prozessrechentechnik, Automation und Robotik
Kaiserstr. 12
76128 Karlsruhe
Germany
+49 (0)721 608 8140
E-mail: yigit@ira.uka.de

Abstract: This paper describes the architecture of our open VR system Visum which offers close to reality testing of mechatronic systems[1]. In addition to the standard functionality of VR systems, like navigation in 3D space, the following points are essential. In order to provide a realistic mechanical behaviour the system has to simulate kinematic and dynamic effects, it has to resolve collisions and the embodied effects of friction and impulse transfer. The characteristics of real mechanical axis drives and servos have to be emulated. The system must also provide complete interfaces to support the execution of real controller software; this methodology is termed software in the loop. We also present a new impulse-based algorithm for the dynamics simulation of linked rigid body systems. A solution for collision detection and resolving can easily be integrated. Interfaces to third party software are provided via CORBA and MCA2. Visum has been tested in several experiments and compared to existing mechatronic systems, e.g. a real robot arm with its controlling software.

Key words: VR-system, mechatronics, dynamics simulation, software in the loop, humanoid robots.

1- Introduction

In technical complex scenarios it is common to test the system behaviour in a simulation environment. Compared to the test procedures on real systems expenses are saved. The simulation is used to achieve qualitative information of the behaviour and the functionality of the system and subsystems respectively. Even in an early development phase when the real hardware still does not exist system optimizations can be realized.

A simulation environment for humanoid robots with the possibility for the user to interact and cooperate with the virtual robot includes many different disciplines, like real-time computer graphics, dynamics and devices for human interaction.
The complexity of such a simulation environment demands a modular system architecture, e.g. complex mechatronic systems such as robots must be built up of simple basic components. Therefore basic components like servos and sensors must be rebuilt and simulated in software. To achieve a realistic behaviour of mechatronic objects in the VR-world a dynamics simulation is required. This is only reasonable when the dynamics simulation incorporates friction, collision detection and collision response. The system must also be capable to integrate third party software via adequate interfaces, e.g. robot control software (software in the loop).
This can only be achieved with an open, extensible and flexible platform.

The outline of this paper is as follows. In the next section we give a short overview of two tools for humanoid simulation systems comparable to our own. We introduce the larger project where our system is part of. This will be followed by a more detailed description of the system architecture and its interfaces to third party software. In the succeeding section we present our approach the dynamics. In the next section we show the results of experiments made with our system controlling a virtual robot arm in comparison to a real robot arm. We close with a short conclusion and an outlook for future work.

2- Related works

Kanehiro et al. [1] describe an open architecture humanoid robotics platform (OpenHRP). OpenHRP is a virtual platform for the investigation of humanoid robots. It consists of

[1] Research funded in part by the DFG = German Research Foundation.

different modules and each is implemented as a CORBA server to obtain a distributed system. For the simulation a robot model, a controller and a client is needed. The model defines the geometry of the robot and is usually given as a VRML model. The controller is a CORBA server implemented by the user which determines the behaviour of the robot. The client is needed to control the simulation. OpenHRP already provides a ready-made client with 3D-graphics and 2D graph display called ISE (Integrated Simulation Environment).
Kuffner et al. [2] describe a large-scale software simulation framework for the development and testing of robot control and behaviour software. The goals of their simulation system are to visualize the robot and its movement in a virtual 3D world, to provide a testing environment for the development and evaluation of robot control and behaviour software, to serve as an interactive user interface for controlling a real robot and to help the robot to make decisions by simulating the consequences of its actions.

3- Background

In our research context we create a VR-system for testing arbitrary mechatronic systems like humanoid robots. All controlling software running on the real hardware must be able to be integrated in our VR-system without any changes. An additional demand is to immerse the user to enable him on the one hand to interact with the VR-world and on the other hand to be recognized by virtual sensors in the VR-world.
We develop this VR-system in the context of the SFB 588 "Humanoid Robots"[2] to provide our partners with a testing environment for their controller, image and speech processing software.

To cope with the complexity of such a simulation environment we develop a modular system architecture. The following essential components must be covered by our system:

- dynamics simulation,
- interfaces for third party software,
- scene description,
- visualization,
- human interaction and feedback.

Complex mechatronic systems such as robots are composed of basic components. These basic components are rebuilt and simulated in software. They have a graphical representation and parameters necessary for a dynamics simulation including friction, collision and collision response. For our research target we need virtual servos and sensors. For the servos we provide standardized interfaces to connect controller software both our own and the software from the robot developers. We support sensors such as cameras, distance sensors and so on.

Such a system needs a special script language to specify the entire VR-world. This includes the graphical description like the geometry, the appearance with colour and texture, point of view and lighting. To describe mechanical properties we provide parameters like mass, inertia tensor, centre of gravity and initial state of motion. Also properties for the objects surface are provided, like coefficients of friction and restitution. To build a robot arm, objects are linked together via different types of joints. At the location of such joints, servos can be placed to achieve an appropriate robot arm. As parameters they have a maximum torque, desired angle and angular speed. They return the actual angle and angular speed.

To give the user a three dimensional visual feedback a powerful 3D vision system based on passive stereo projection is used.

To interact in the VR-world the user is tracked to control his avatar in the VR-scene. With the virtual cameras the robot watches the gestures of the human being, respectively, his avatar. Then the robot can react appropriately to a recognized gesture. With data gloves the user is able to cooperate with the robot, e.g. carrying of a table.

In the following, we describe how we solved the requirements defined in this section.

4- System architecture

The Visum architecture incorporates nearly all of the above mentioned features like the dynamics and graphical models for a virtual environment. The ability to connect the user to the system in order to interact in the virtual world has not been fully implemented yet. Developing our system we took the following list of common demands into account:

- easy extensibility,
- open API,
- easy to debug,
- easy to handle,
- easy integration of robot control software.

To have a platform independent system we use Python instead of a low level programming language like C/C++. With the high level language Python we have a good foundation for a flexible and easy to extend system. Only the aspect of real-time is at odds with Python. In some cases it is much slower then C/C++. But Python offers several options to speed up programs, e.g. the time critical code is written in C/C++ and wrapped via Pyrex[3] to be accessed from Python.
The actual VR-system Visum is fully implemented in Python using a predefined set of abstract base classes. In the derived classes the proper functionalities are implemented. They are integrated in the system as Plug-ins to keep the core program untouched.

[2] Homepage of the SFB 588: http://www.sfb588.uni-karlsruhe.de.

[3] Other wrappers are SWIG, Sip and Boost.

The Visum base classes are:

- *Scene:* acts as a container for the VR-world objects.
- *Node:* all objects which can be stored in the scene are from the type *node*.
- *Controller:* a controller transforms a node. This is where the dynamics is located.
- *Renderer:* displays the scene on the screen (e.g. via OpenGL).

There are also additional classes supporting a base interaction in the scene (dolly, translate, rotate) which are not mentioned here. Distinguished nodes in the scene are:

- *Agent:* these objects possess an own logic for their behaviour.
- *Sensors:* they supply the agents with input data.
- *Actuators:* the drive of an agent.

The simulation only takes place in a controller object. Every controller can modify an arbitrary number of nodes. This enables the parallel use of different simulation methods in a single VR-world. Actually, we support two simulation methods. These are our own method and the Open Dynamics Engine ODE [3]. Nodes without a controller are just static scenery.

The cycle of a single simulation step is portrayed in figure 1. The first three steps concern the sensors, the agents and the actuators. Afterwards every controller is executed and the nodes receive their new position in the scene. Finally the renderer produces the image of the scene. To render the scene in Python we use OpenGL. It is fast enough, because the OpenGL API is directly mapped to Python and the geometry is drawn using OpenGL's display lists.

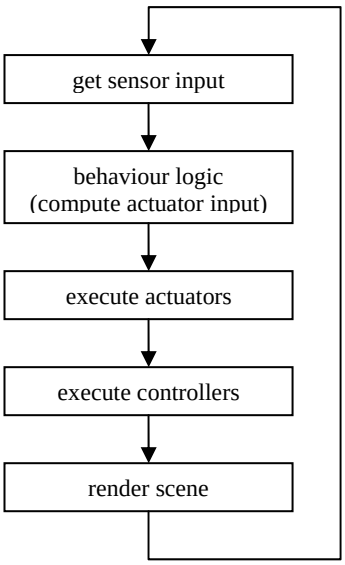

Fig. 1 : Cycle of a single simulation step.

At the moment our scene description is given in pure Python code. This is highly versatile but not useful for non-experts. Therefore we design for future purposes a human readable and easy to use scene description

Due to the high flexibility of Python, even during runtime, scene nodes can receive new attributes and methods, e.g. a controller object may add attributes to its nodes needed for the simulation, like friction parameters.
Software can be integrated in Visum via controllers. In section 6 we give an example of how we attached a controller software for an Amtec [4] arm via MCA2 (Modular Controller Architecture Version 2) [5]. MCA2 is a modular network transparent framework for controlling robots and other kinds of hardware, developed at the FZI[4] in Karlsruhe Germany.
With this feature third party software can be tested in our virtual environment. Because the controller software uses MCA2 the real hardware can be easily substituted by the virtual hardware. Therefore, the controller software runs without changes on the real as well on the virtual hardware. We describe this technique as software in the loop. Integrating an entire robot in our VR-environment will result in a robot in the loop simulation.

To allow immediate visual feedback of the 3D situation and not to restrict the moving environment, the user stands in front of a rear projection screen using polarised glasses (see figure 2).
To enable communication and interaction between the "real" user and the "virtual" robot it is necessary to integrate the 3D model of the human shape in the "virtual" vision system of the robot.
Normally this could be done by electromagnetic motion capture systems. But the exorbitant prices and the various restrictions (extensive calibration, limited moving area, no image of the surface, etc.) forced us to look for more effective solutions. Currently we investigate a camera based system which uses the concept of visual hulls introduced by [6]. Eight calibrated cameras surround the "real" user. Out of these 8 images the human silhouette is calculated. The intersection of these silhouette cones defines an approximate geometric representation of the human shape called the visual hull. In a final step this polygon model is textured in real-time and added as avatar to the VR-world. Off-line experiments with captured images of static situations promise refresh-rates near to real-time [7].

[4] FZI: Forschungszentrum Informatik, http://www.fzi.de

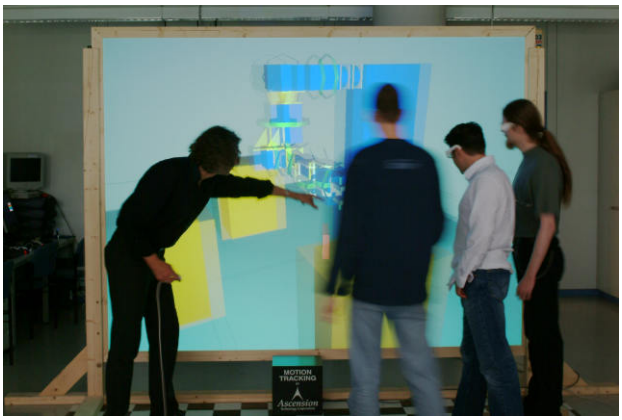

Fig. 2 : Power wall.

5- Impulse-based dynamics simulation

We will now describe our new dynamics simulation algorithm in some detail. Our algorithm demonstrates that the well-known impulse-based collision resolving algorithms (e.g. Mirtich and Canny [8]) can be generalized and extended to deal with interconnected rigid body systems. The goal of this presentation is to point out the basic principle of our new algorithm. A discussion of its strengths and weaknesses is presented at the end of this chapter.

All forces we will deal with are impulsive forces or impulses. If you integrate Newton's second law $F(t) = m \cdot \dot{v}(t)$ the continuous force $F(t)$ in the time interval $t = 0 \ldots h$ is transformed into an impulse I with no time parameter t any more:

$$I = \int_0^h F(t)dt = m\int_0^h \dot{v}(t)dt = m(v(h) - v(0)) = m \cdot \Delta v$$

An impulse I applied to a mass point m changes the velocity instantaneously by $\Delta v = I / m$. This concept is easily transferred to torques.

Definition: A system of n linked rigid bodies is defined by the following parameters

(1) $m_k \in R$ the total mass of body $k = 1 \cdots n$,

(2) $C_k(t) \in R^3$ its (possibly moving) center of gravity,

(3) $\dot{C}_k(t) = \frac{dC_k(t)}{dt} \in R^3$ its velocity,

(4) $\omega_k(t) \in R^3$ its angular velocity or spin,

(5) J_k its 3x3-matrix of inertia (referring to body space coordinates),

(6) R_k its orientation matrix mapping vectors from body space coordinates to world coordinates.

What is still missing are the interconnections between the rigid bodies. To simplify this short presentation, we only consider spherical joints. In real life numerous other types of interconnections are needed, e.g. spring and damper elements, revolute joints, prismatic joints, screw joints, cylindrical joints and points sliding on planes and curves etc. We have not yet implemented all types of joints, but as far as we know until now, our simulation method allows a wide variety of joints including all the above mentioned ones.

Definition: (continued ...)

(7) Attached to each body is a list of points $P_{ki} \in R^3$ fixed on this body and moving with the body:

$$P_k = \{P_{k1}, \ldots, P_{ki}, \ldots\}$$

(8) The list of all spherical joints is given by a set of 4-tupels of indices

$$L = \{\ldots, (k_1, i_1, k_2, i_2), \ldots\}$$

where k_1, k_2 denote 2 different bodies and $P_{k_1 i_1}$ and $P_{k_2 i_2}$ are two points fixed on the respective bodies.

A joint (k, i, k', i') thus defines a pair of points where two rigid bodies are connected or fixed to each other. Thus a joint of this type has three degrees of freedom, whereas a revolute joint with one degree of freedom can be modeled with 2 joints of the type introduced above. For our method it is of no importance whether there are open or closed kinematic chains. An intermixture of both is allowed.

The global strategy of our simulation algorithm is as follows: At time t we assume to have a consistent state of the n-body system. That implies, that the actual positions of the bodies in world space fulfill the distance condition

$$\forall (k, i, k', i') \in L : \left| P_{ki}(t) - P_{k'i'}(t) \right| \le D\max .$$

where Dmax is a small constant, e.g. $D\max = 10^{-6}$.
We do not postulate the equality of the velocities of these points, i. e. $\dot{P}_{ki}(t) = \dot{P}_{k'i'}(t)$, for reasons that will be explained later.

Our dynamics simulation method does never use continuous forces but impulses. This implies that a time step from t to $t+h$ is executed without continuous inner forces. The bodies instead move on ballistic curves. The continuous inner forces are substituted by suitable impulse changes at time t. But how can we calculate the correct impulses to end up at time $t+h$ in a consistent state? It is done iteratively and this is probably the most innovative detail of our new algorithm. We use a look-ahead function

$$P_{ki}(t+h) = Integrate(k, P_{ki}(t), h)$$

that simulates the motion of body k on ballistic curves inside the time step h. *Integrate* is not a simple function. In order to calculate the position of point $P_{ki}(t+h)$, we have to solve the equations of motion of the rigid body with subscript k in the time interval $[t,t+h]$. This unconstrained motion of a single rigid body in the absence of external forces is described by the well-known Euler equations (in body space coordinates)

$$\dot{\omega}(t) = -J^{-1}(\omega(t) \times (J \cdot \omega(t)))$$

for the rotational part and by Newton's second law for the motion of the center of mass. For the solution of these differential equations an elegant solution up to any degree of precision is possible with the Taylor series method. J is the inertia tensor of the body and it is advisable to use the principal axes of inertia as directions of the body-fixed coordinates. In this case, only the diagonal elements J_{11}, J_{22}, J_{33} are possibly nonzero. We calculate the inverse by setting the diagonal elements to $1/J_{11}, 1/J_{22}, 1/J_{33}$, and if one of the J_{ii} is zero, it remains so.

The iterative procedure for the calculation of the impulses substituting the inner forces operates as follows:

Given a *consistent state* of the linked rigid body system at time *t*.

While there is a spherical joint $(k, i, k', i') \in L$ with

$$Distance = \left| P_{ki}(t+h) - P_{k'i'}(t+h) \right| > D\max$$

Do: $delta := P_{ki}(t+h) - P_{k'i'}(t+h)$; Determine a correcting impulse F (details explained later) and change the angular velocities by

$$\omega_k(t) := \omega_k(t) + \tilde{J}_k^{-1} \cdot ((P_{ki}(t) - C_k(t)) \times F),$$

$$\omega_{k'}(t) := \omega_{k'}(t) + \tilde{J}_{k'}^{-1} \cdot ((P_{k'i'}(t) - C_{k'}(t)) \times (-F))$$

and the velocities of the respective centers of gravity by

$$\dot{C}_k(t) = \dot{C}_k(t) + F / m_k,$$

$$\dot{C}_{k'}(t) = \dot{C}_{k'}(t) - F / m_{k'}$$

End

Remarks: The matrices $\tilde{J}_k^{-1} = (R_k \cdot J_k \cdot R_k^T)^{-1}$ are the inverses of the inertia matrices in world space coordinates. The look-ahead point locations $P_{ki}(t+h)$ and $P_{k'i'}(t+h)$ are calculated by the function *Integrate*. With the updated parameters of the two bodies we normally will have $\left| P_{ki}(t+h) - P_{k'i'}(t+h) \right| \le D\max$. If not, the while loop will repeat until all joints are corrected. Is the new state of the body system still consistent after a correction is done inside the while loop? The answer is yes, since increments of impulses only change velocities, not positions.

For the calculation of the distance correcting impulse F we use the formulae

$$r_1 = (P_{ki}(t) - C_k(t)),$$

$$r_2 = (P_{k'i'}(t) - C_{k'}(t)),$$

$$\Delta v_1 = (\tilde{J}_1^{-1}(r_1 \times F)) \times r_1 + F / m_k,$$

$$\Delta v_2 = (\tilde{J}_2^{-1}(r_2 \times -F)) \times r_2 - F / m_{k'}$$

$$\Delta v_1 + \Delta v_2 = delta / h.$$

Since F is the only unknown term, the last line is in fact a linear system of three equations that gives us a good approximation for the unknown *F*. The idea behind this is to close the gap *delta* at time *t+h* between the two joint points by a suitable change of the velocity increments Δv_1 and Δv_2 at time *t*. A small problem is left. We calculate *F* at time *t* and do not include the influences by nutational changes of spin axes, whereas the function *Integrate* does include nutation. This small deviation is of no practical importance since corrections at one joint normally disturb the distance conditions of others, but in the future of the iterative process the deltas will be smaller and smaller. We have never seen a mechanical model where this iteration does not converge, if the step size h is choosen small enought.

When the while loop terminates, we have reached such a state, that we can advance to time t+h. We use the function *Integrate* and Newton's law to do this properly. The correcting strategy of the while loop including look-aheads guarantees that the new system state at time t+h is consistent. The correcting impulses F substitute the continuous inner forces at the joints.

But what about the velocities? Immediately after entering the new state at time *t+h* we normally observe $\dot{P}_{ki}(t+h) \neq \dot{P}_{k'i'}(t+h)$. These undesirable differences of velocities of joint points can be eliminated. We use essentially the same structure of the while loop as described above but this time without a look-ahead. We finally end up with

$$\forall (k, i, k', i') \in L : \left| \dot{P}_{ki}(t+h) - \dot{P}_{k'i'}(t+h) \right| \le V_{\max}$$

where $V_{\max}$ is a suitably small constant, say $V_{\max} = 10^{-4}\, m/s$. Experience with the simulation of numerous linked rigid body models has shown, that the velocity correction as described above is only of a cosmetic nature. It has no measurable influence on the point positions at time $t + i \cdot h$, $i = 1, 2, \ldots$ and is thus only of use if accurate values for velocities are needed. But, after this remark, it should be mentioned that the velocity correction by impulsive forces at a fixed time is exactly that algorithm that is needed to resolve collisions of linked rigid body systems, see e.g. chapter 6 of Wittenburg [9].

Our experience with the new iterative impulse-based dynamics simulation method so far can be summarized as follows: We can compute very accurate solutions or, if we relax the parameter $D\max$ and choose $h = 0.04s$, simulations can be made very fast. Our new algorithm is a so-called "anytime" algorithm. Even if in real-time environments the simulation is under extreme time stress and the number of correction steps of the while loop for each joint must be reduced to 1, mechanical models do not disintegrate but are damped a little bit. On a contemporary PC with 2 GHz, about 200,000 while loops are executed per second.

If h and $D\max$ are made smaller and smaller, the calculated solution converges to the exact physical solution of the linked rigid body system. The error is of the order $O(h^2)$. If $D\max$ is reduced, e.g. to values of 10^{-6}, energy conservation is nearly perfect even for mechanical models with a chaotic behaviour. Since we do not use systems of differential equations as is the case for other dynamics simulation software known to us, all the pre-processing is of a trivial nature and this is also true for our main iterative while loop.

All this qualifies our simulation method for implementations

in future VR systems with fully integrated rigid body dynamics. Collision detection and collision resolving as described e.g. by Mirtich and Canny [8] and others can be included easily. Since all our forces are of an impulsive nature, it is easy to integrate various collision laws and Coulomb friction including slipping. We will report on these matters in forthcomming publications.

6- Experiments

In this section we present three example scenarios that have been simulated in our experimental VR environment. The first two are just examples of simple robots having servos and sensors which enable them to move around and explore their environment. The third example is a simulation of a real robot arm which is made of a couple of Amtec modules and is located at one of our partner institutes, namely the *Institut für Prozessrechentechnik, Automation und Robotik* (see figure 5).

In figure 3 you can see a screenshot of Visum running a simulation of two simple robots. This example shows two kinds of locomotion. In the background you see a cart that is driven by wheels and in the foreground there is a robot that moves by crawling across the floor. Every movement of both robots is driven by the dynamics simulation. The simulation of friction enables the robots to move around.
The car has four wheels which are attached to the chassis by simulated joints. The joints have motors that can be velocity controlled. The front wheels have an additional degree of freedom which can be used to steer the cart.
The other robot has an arm made of three segments that are connected with revolute joints which are position controlled. By stretching and flexing this simple arm and scraping on the floor the robot is able to pull itself forward.

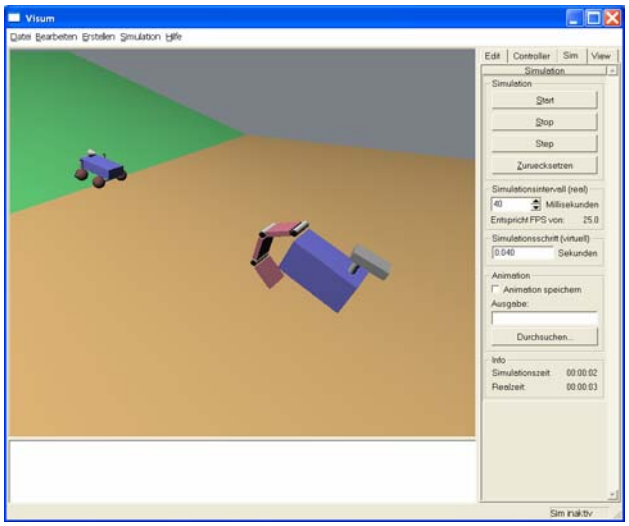

Fig. 3 : Two simple robots moving in a virtual environment.

In the second example (figure 4) the cart from above was extended and got a distance sensor attached to its chassis. This sensor casts a ray in a particular direction to measure the distance to the next obstacle. Using this sensor the robot can actually explore its environment and avoid running into obstacles or against walls.

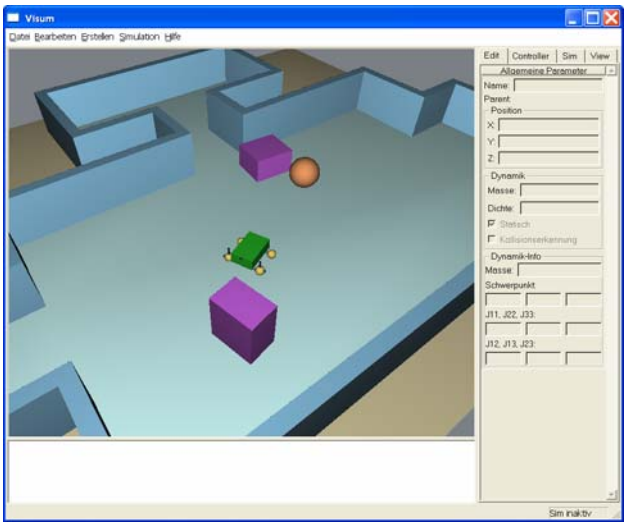

Fig. 4 : A robot cart exploring its environment.

The next example is a bit more involved, it demonstrates the simulation of a robot arm which is controlled by the very same software that is also used for the real robot arm. Additionally, the controlling software and the VR environment are executed on two separate machines.
The real robot arm is an assembly of several Amtec PowerCube modules (see figure 5). The arm itself has 7 degrees of freedom and can be controlled by several modes. The controlling software builds on top of the MCA2 framework.

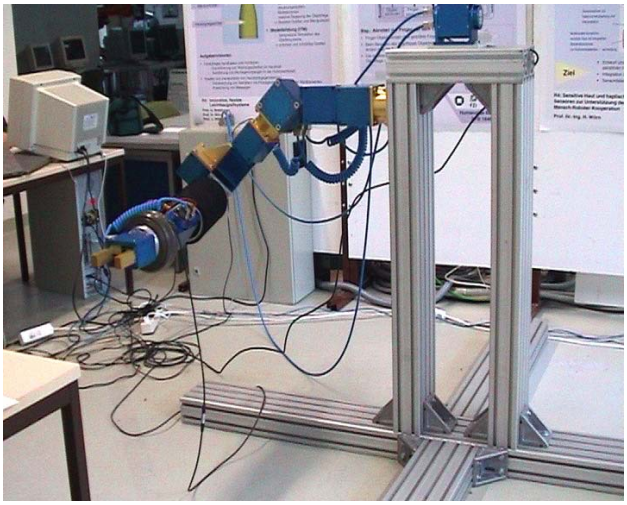

Fig. 5 : The real Amtec robot arm.

In MCA2 you break down your controlling software in small functional chunks called *modules* that all have the same basic structure and that encapsulate a specific task that can be reused in other controllers. Such modules can then be connected with each other forming a hierarchy. They can also be combined into *groups* or *parts*. The basic structure of modules, groups and parts is always the same. They receive controller input from above, do their internal processing and pass their controller output values on to other modules. In the

other direction they receive sensor input from below, do again some internal processing and pass sensor output up to whatever module they are connected to. Furthermore, they can have internal parameters and variables necessary to accomplish their respective task (see figure 6). Any of the controller, sensor or internal data can be inspected or modified by special tools at runtime which helps testing or debugging the controlling software.

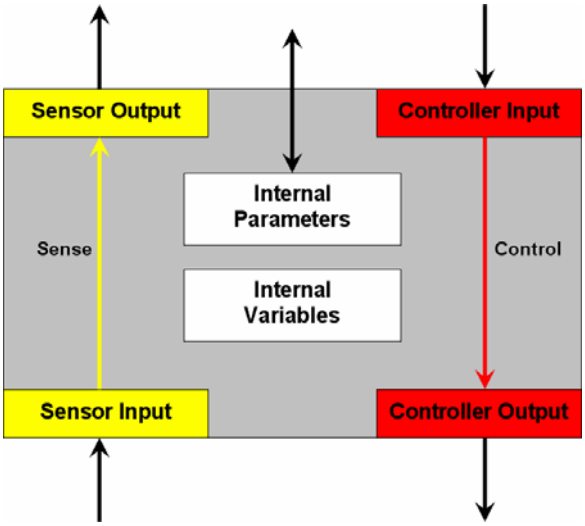

Fig. 6 : Structure of a MCA2 module.

At some point in the MCA2 hierarchy of the original controller software there is a MCA2 part that directly controls the actual hardware. This part receives the joint angles and maximum speeds as controller input and provides the current joint angles as sensor output. In order to do a simulation this particular part has to be replaced by a part with the same interface but that does not control the real hardware but the virtual hardware. All other modules remain the same. MCA2 already allows such a replacement even without recompiling the programs. You could even connect both parts at once and control the real robot and the virtual robot at the same time. Of course, only one of them can actually be part of the control loop. And as MCA2 can transfer data also via a TCP connection, the simulation part can even run on a different machine than the actual controlling software. By using these mechanisms it is easy to switch between the real robot and the simulation. In the simulation the virtual robot arm runs in real-time. It consists of seven hinge joints for the arm and two slider joints for the gripper.
A screenshot of the simulation can be seen in figure 7. The robot model was originally modelled in ROBCAD, imported into 3D Studio MAX where it was prepared for the simulation and finally exported into our own format that is actually just Python source code. This data can then be imported into our VR environment. During the initialization of the simulation our MCA2 simulation part is launched and gets connected to the controlling software. Now everything is set up to receive the joint angles and maximum speeds of the original controlling software and forward them to the VR environment where it will drive the virtual robot arm.

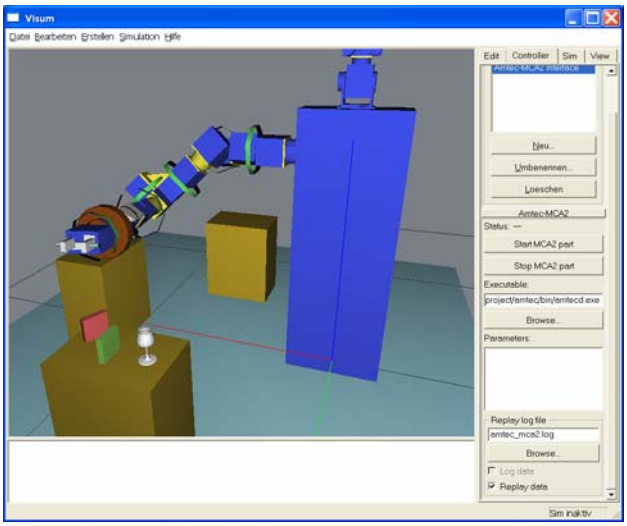

Fig. 7 : The simulated Amtec robot arm.

7- Conclusion and future work

In this paper we introduced Visum, a VR-system for the interactive evaluation of complex mechatronic systems. In the description of our research background we outlined the scope of our work and defined the requirements. Afterwards we explained the system architecture and how it was designed to meet the given requirements. Then we presented our new algorithm for dynamic simulations. In an experiment where we compared a real and a virtual robot arm we demonstrated that our system is capable of integrating third party controller software on our virtual hardware.

As we mentioned above, the immersion of the user is not integrated yet. Based on our successful off-line experiments with captured images of static situations we will rearrange this software to work in real-time with eight cameras generating a 3D model of the user.
To enable non-experts to work with our system, we design a human readable and easy to use scene description language including graphics and dynamics. Thereby our design is oriented by common languages such as VRML, POVRAY and so on. The script is based on a semantic description and has different encodings like X3D. For easy integration in Visum we will have a Python-encoding. To be able to manipulate the script with existing editors and modellers we will include a VRML-encoding.

Acknowledgement:
The authors would like to thank J. Wittenburg for his engaged and valuable advice on dynamics. Thanks go also to J. Bender, St. Preuß and G. Stelzner for their close cooperation within the Visum project.

8- Bibliography

[1] Kanehiro F., Fujiwara K., Kajita S., Yokoi K., Kaneko K., Hirukawa H., Nakamura Y., Yamane K. Open Architecture Humanoid Robotics Platform. In Proc. of IEEE International Conference on Robotics and Automation (ICRA2002), Vol.1,

pages 24-30, Washington D.C., USA, May 2002.

[2] Kuffner J.J., Kagami S., Inaba M., and Inoue H. Graphical simulation and high-level control of humanoid robots. In Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS2000), Vol. 3, pages 1943-1948, Takamatsu, Japan, November 2000.

[3] Smith R. Open dynamics engine. http://opende.sourceforge.net. June 2003.

[4] Amtec robotics GmbH. http://www.amtec-robotics.com. June 2003.

[5] Scholl K.-U. MCA2 - Modular Controller Architecture Version 2. http://mca2.sourceforge.net. June 2003.

[6] Laurentini A. The Visual Hull Concept for Silhouette-Based Image Understanding. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 16, No. 2, February 1994.

[7] Fautz M. Objekt und Texturrekonstruktion mit einer robotergeführten Kamera. Dissertation, Universität Karlsruhe (TH), Fakultät für Informatik, October 2002.

[8] Mirtich, B., Canny J. Impulse-based Simulation of Rigid Bodies. Symposion on Interactive 3D Graphics, Monterey, Cal., April 1995.

[9] J. Wittenburg. Dynamics of Systems of Rigid Bodies. B. G. Teubner, Stuttgart 1977.