Generalized Object Interactions in a Component based Simulation Environment

Tom Van Laerhoven Chris Raymaekers Frank Van Reeth
Expertise centre for Digital Media
Wetenschapspark 2
Diepenbeek, Belgium
{tom.vanlaerhoven, chris.raymaekers, frank.vanreeth}@luc.ac.be

ABSTRACT

We describe the procedure for generalizing the representation of interacting virtual objects in a component based simulation framework. This extends our previous work where we did the same with the representation of the objects themselves. Isolating the interaction mechanisms into separate components provides us with several advantages; one of them is the ability to replace an interaction component by another, possibly at run time.

An example simulation scene contains three kinds of interacting rigid bodies, using collision detection and response, simple velocity constraints, and scripted interactions. The example shows an engine that drives two gears and a fan. The generated airflow from the fan causes a balloon to hit another balloon.

The goal of our work is to create a flexible an extensible "tinker toy" environment that incorporates different simulation domains while reusing existing tools.

Keywords

Physically based simulation, rigid body, constraint, framework, virtual environment

1. INTRODUCTION

Handling the physical interactions between the participating objects in a virtual environment is a challenging task. The vast number of different kinds of interactions, each of which can be treated using a whole set of available techniques, makes it hard for developers of simulation environments to choose the right algorithm in a particular situation. This choice however is important, because it has effect on both the physical and visual correctness of the simulation, and its real-time properties. Typically, a trade-off between these aspects has to be made prior to implementation, and it is very hard or even impossible to make changes afterwards. Allowing such changes to the interaction mechanisms can be desirable however, the more if these can happen at run time, like we will see further on.

Creating a flexible environment that handles the above issues is difficult for several reasons. First of all, objects from disparate simulation domains each require specialized simulation techniques. For example, simulating particles means supporting large quantities of simple objects with simple or no interactions. Cloth objects must cope with complex deformable surfaces and the resulting stiff equations that need to be solved with suitable differential equation solvers. In the same way, the interaction mechanisms that describe how objects communicate with each other have to be treated with attention to their computational needs. These include the interferences between rigid bodies, which involves complex operations like detecting collisions and resolving contacts, interaction between parts of an articulated body and the response to external forces in general.

Combining objects from disparate domains into a single simulation makes things even more complicated. This not only means supporting the different needs of various simulation domains, but also the cross-domain interactions, which are far more problematic. Most environments are dedicated and target only a single domain, not allowing for extensions to other domains or different interaction methods.

This paper does not address all of the above issues but describes our approach in dealing with

Journal of WSCG, Vol.11, No.1., ISSN 1213-6972
WSCG'2003, February 3-7, 2003, Plzen, Czech Republic.

some of them, along with some early results. By obtaining a certain level of abstraction where both virtual objects and their interaction mechanisms are described as sets of components, we get a flexible and extensible environment. At the same time, this abstraction opens up the possibilities for cross-domain interactions. It also creates the interesting ability to replace interaction algorithms at run time, creating a level-of-detail approach for interaction algorithms in simulations. This can be useful in virtual environments that target visual correctness, rather than physical correctness.

The rest of this document is organized as follows: the next section gives an overview of related work in the domain of physically based simulations. Section 3 gives a brief description of our architecture that provides us with the means to achieve our goals. Next, in section 4 we will elaborate on the approach of abstracting the interactions themselves, followed by some examples in section 5. Section 6 presents the next steps in this work. Finally, the conclusions about our work are drawn in section 7.

2. RELATED WORK

In rigid body dynamics, modeling the interactions between interfering objects is the most difficult part of the simulation. Two major approaches that try to solve this problem can be identified: the constraint based methods and the impulse based approach.

An overview of the use of constraints to avoid interference or penetration is given in [Bar97b]. In general this means restrictions on the way the objects are permitted to move are formulated. This can be done using energy functions that act as generalized spring forces, better known as “the penalty method”, or by converting object's accelerations into “legal” accelerations using constraint forces.

A departure from these traditional constraint-based methods came when Mirtich introduced “impulse-based simulation” [Mir96]. In this approach constraint forces are no longer explicitly applied between two contact points, but contacts are exclusively modeled through collision impulses that are applied between the interfering bodies. [Mir96] also addresses the issue of hybrid simulations, combining both types of simulation paradigms.

Apart from solving the problem of interfering objects, there is also the concept of simulating linked articulated rigid bodies. In this case geometric constraints are posed on objects, linking them together. These systems are traditionally solved by reformulating the constraints as algebraic equations or by a technique called degrees of freedom analysis [Kra90].

Instead of objects interacting among each other, there is also the issue of users interacting with objects in a virtual environment. [Wit90] presented a formulation for constrained dynamics that makes it possible to dynamically create complex physical models by snapping simple building block together. This way the process of model creation is integrated while running the simulation. Also related to this subject, [Smi01] describes a construction system that restricts object interactions, based on human intuitions, and automatically generates constraints for geometric objects.

In [Bar97a] the need to incorporate different simulation domains within one simulation environment was pointed out. Using a technique called interleaved simulation, the authors treated several existing simulators from different domains as black boxes with simple generic interfaces. They combined a cloth simulator with a rigid body simulator, and a particle system with a rigid body simulator. This was done by instructing each system to take a step, first without regard to other system's constraints, then taking another step consistent with the already computed motion of other systems.

3. A FRAMEWORK FOR PHYSICAL SIMULATIONS

The pLab project, introduced in [Lae02], consists of a plugable component framework that aims on building highly extensible physical simulation environments. It accomplishes this by using plugins that introduce building blocks in the form of components into an application. An application built upon the framework uses an XML (eXtensible Markup Language) document [W3c02] as a blueprint of the component layout. This procedure is explained in more detail further on in this section. The motivation here was to find a way to easily combine and exchange existing tools in the domain of physically based simulation.

Since we will continue and extend our work on the pLab framework in this paper, this section will give a brief overview of its features.

Abstracting Simulation Objects

The work described in [Lae02] explains the process of abstracting a simulation object in the pLab framework as a collection of separate components. Adding or exchanging components can alter the behavior of an object.

This procedure treats individual objects in an environment as black boxes, all of which can be

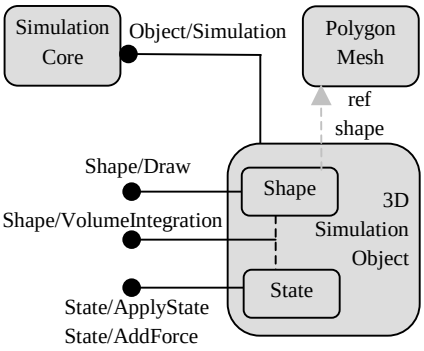


Figure 1. An abstract view on a dynamic simulation object and some of its properties.

accessed in the same way through a generic interface. The approach is different from the one proposed in [Bar97a] in the way that instead of whole simulators, individual objects are treated as black boxes with their own interface.

As an example, a simple static object in an environment would be described by a shape component containing all the object's visual properties. Adding a simple state component would give it a place and orientation in the environment. Replacing the state component with a more complex version, which keeps track of dynamic properties like velocity and the ability to receive forces, would let the object exhibit some dynamic behavior.

In the same way, the object can be given the ability to be human-triggerable by adding a user interaction component, containing the information on how it is to be triggered and what the resulting action would be. Continuing this approach, a complete functional 2D or 3D user interface can be built [Luy02].

Figure 1 shows a schematic description of a typical simulation object in the framework. The object has a shape component, which in this case is defined by a reference to a polygon mesh. Other objects with a similar shape can refer to the same component to define their visual properties. The state component of the object contains all properties to make it a rigid body.

Components can also export operations, called "commands". For example, the object is rendered on screen by calling the *Draw* command, and the *VolumeIntegration* command couples state and shape components by calculating the mass properties and setting them in the state component. The "Simulation" command describes the object's actions in each of the different simulation stages. For example, the "Step" stage typically calculates the object's state in the next time frame by integrating its properties. The rendering of the object takes place in the "PostStep" stage. Placing this simulation command in the simulation core ensures that the object participates in the environment.

The XML Blueprint

The components, which together form the application, are all introduced by separate plugins. Each plugin can be seen as a supplier of building blocks or components that have related tasks. For instance, the shape plugin contains several different kinds of shape components and their associated commands. The difference with traditional plugins is that traditional plugins have a fixed and limited functionality. In our case the role of the plugin's contents is defined at run time, when the components are coupled. So in addition we need a description of how these components in the application are related to each other. This is realized using an XML document.

The listing in figure 2 shows an example of such an XML document. It describes part of the same component layout as figure 1. Component tags can use a type attribute, containing a service name, to indicate what kind of component they want to embody. The service name is just an expression that associates certain functionalities with the component. The id attributes create a mechanism for referencing other components. The commands are created in the same way, but additionally have a subject tag to indicate on which component they operate.

```
…
<component id="00" name="Dummy Mesh">
   <!-- Embedded XGL code -->
   <MESH>...</MESH>
</component>

<component id="01" name="Dummy Object">

   <!-- Component's body -->
   <component id="02" name="Dummy Object Mesh">
      <compref>00</compref>
   </component>

   <!-- Component's interface -->
   <command id="08" subject="02" type="Shape/Draw"/>
   ...
   <!-- Simulation command -->
   <command id=”09” name=”DummyObject/Simulation”>
      <command name=”Stage/Init”>…</command>
      <command name=”Stage/PreStep”>…</command>
      <command name=”Stage/Step”>…</command>
      <command name=”Stage/PostStep”>…</command>
      <command name=”Stage/Rewind”>…</command>
      <command name=”Stage/Deinit”>…</command>
   </command>
</component>
...
```

Figure 2. An extract from a XML document, describing a virtual object.

One of the advantages of using XML here is the fact that existing XML formats can be reused by embedding them into the component descriptions. For instance, shapes can be defined by using XGL [Xgl02], an XML format designed to capture the 3D information of object geometries that can be rendered by OpenGL [Woo99].

The framework supplies an XML parser, the Expat parser [Cla01], which is wrapped into a plugin and to which XML handlers for various formats can be attached. A handler for the XGL format is also provided.

4. ABSTRACTING OBJECT INTERACTION MECHANISMS

Section 3 explained the approach of representing a virtual object as a set of components, each of which contributes to the object's properties. It allows the introduction of disparate objects in a single environment, all having the same underlying abstract representation. However, it does not take into account the interactions between objects. These include interactions caused by complex mechanisms like collision detection and response, the joints in an articulated body, non-contact forces like gravity and magnetism, but also user interactions.

Generalizing these mechanisms between objects in the same way as we did with the objects themselves, by describing them as sets of components, provides us with the following advantages:

- The ability to combine different interaction mechanisms that have different needs into one simulation, all having the same representation.
- The use of scripted interaction mechanisms or ad hoc solutions for specific situations.
- Exchange interaction mechanisms with more suitable versions at runtime.

The possibility to exchange the interaction mechanisms at run time has some interesting applications. Consider a virtual environment that contains a complex simulation. If the user is far away from the simulation scene, or if the scene is positioned behind the user, simple and computationally efficient algorithms are used to let the objects in the scene interact. If the user comes closer however, the simulation switches to algorithms that provide more visual accurate results.

Because these algorithms are now isolated into a separate interaction component, this approach becomes feasible.

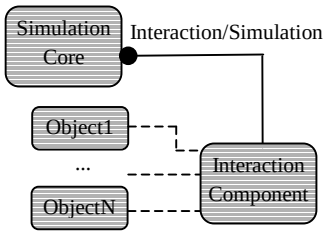


Figure 3. A schematic view on an interaction component between N objects.

Description of the interaction component

Figure 3 depicts an interaction component that handles the communication between N bodies in an environment. Similar to the description of the objects themselves, a simulation command contains the operations that should be executed in each stage of the simulation.

Section 5 shows an example scene in which different interaction mechanisms are combined in one simulation. The rest of this section discusses three different interaction techniques that were used in this scene, along with some example descriptions in XML.

Collision Detection and Response

The field of collision detection is probably one of the most covered fields within the domain of virtual environments. This results in a great amount of collision libraries, which have the complex task of reporting the interferences between several objects.

```
<!-- Instantiate collision library -->
<component id="10" type="CDetection/SOLIDlib">
   <command id="11" subject="10" type="CDetection/Handle"/>
</component>
…
<!-- Interaction component -->
<component id="00" name="Collision  Interaction">
   <component id="01" name="CollisionComponent Body1">
   …
   <component id="0N" name="CollisionComponent BodyN">

   <!-- Simulation command -->
   <command id="01" name="ExampleCollision/Simulation">

      <command name="Stage/Init">
         <!-- Add objects to collision library -- >
      </command>
      <command name="Stage/Step">
         <!-- Update states in collision library -- >
      </command>
      <command name="Stage/PostStep">
         <!-- Handle collisions -- >
      </command>
   </command>
</component>
```

Figure 4. An XML example of an interaction component handling collisions.

Again, the difficulty here is that all libraries target different domains and needs.

A few examples: the H-COLLidE library [Gre99] targets haptic interaction, the V-COLLidE library [Hud97] large environments with many objects, PQP [Got99] supports additional distance computations, and SOLid [Ber99] can detect interferences in an environment that contains deformable objects.

Figure 4 depicts part of the description of an interaction component in pLab that supports the detection of collisions. Instead of defining interaction components between every body that participates in the collision detection and response process, which typically involves nearly all objects in the environment, we define only one interaction component that interacts with the collision library and all participating objects. In the example of figure 4, the library is the SOLid collision detection library [Ber99].

Additionally, we provided an extension in the form of a plugin to define the collision resolution algorithm, which ensures that none of the objects overlap. In this case, the simple but efficient bisection method was used. This procedure possibly involves rewinding the simulator to a previous time frame where no interferences were reported. This means all related objects have to provide actions for the "Rewind" stage of the simulator (figure 2), and have to keep previous states in a *Memento* component.

Another extension defines the behavior of interfering objects after their collisions are resolved. We implemented the impulse-based method proposed in [Mir96], which calculates the impulses that need to be applied to interfering objects in order to break them apart.

Constraints

The collision detection and response technique is not always suitable for the simulation of object interactions. For instance, when a drawer is pulled out of a cabinet, the path that the drawer follows

Figuur 5. Two constrained gears.

```
<!-- Simulation command -->
<command id="01" name="ExampleConstraint/Simulation">

  <command name="Stage/Step">
    <command subject="52" type="Constraint/AngularVelocity"/>
      <cmdparam type="SetTargetState">
        <udata><cmdref>202</cmdref></udata>
      </cmdparam>
      <cmdparam type="SetVelocityFactor">
        <udata>-0.5</udata>
      </cmdparam>
    </command>
  </command>

</command>
```

Figure 6. An XML description of the simulation command of a constraint.

within the cabinet can easily be described using simple geometric constraints. The application of collision detection and response in this particular situation would lead to excessive calculations, and possibly a worse result.

Clearly, in many cases it would be sufficient that the cabinet and the drawer interact using the geometric constraint mechanism.

Geometric constraints are also useful when simulating articulated bodies. These comprise a set of rigid bodies connected by joints, forming a tree, a chain or a graph [Kra90]. Because external forces and other joints influence the forces applied at each joint in the system, possibly creating loops, suitable algorithms are needed to calculate the resulting forces at each joint.

Figure 6 shows an example of a simulation command that enforces the states of two objects to have the same angular velocity. An extra parameter, the velocity factor, can be used to define a linear relationship between two velocities. It will be used in the example scene to express the fact that a gear that is two times the size of another gear, which is connected to the first gear, will rotate half as fast (figure 5). This simulation command is embedded in the interaction component.

At the moment we consider only simple constraints between two bodies. However, the results remain valid if we introduce a constraint manager that resolves complex constraint graphs.

```
<command id="300" name="Hello" type="Script/Python"/>
   <cmdparam type="SetCode">
      <udata>print "Hello pLab!"</udata>
   </cmdparam>
</command>
```

Figure 7. An XML description of a Python script, embedded in a command.

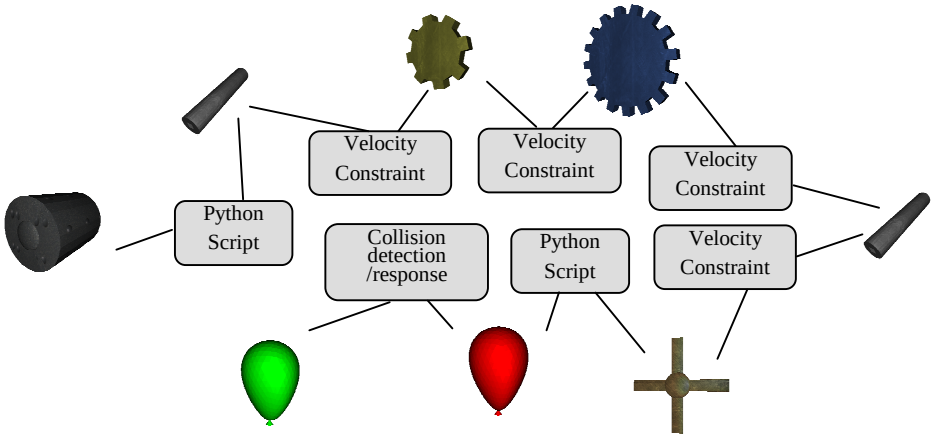


Figure 8. Schematic view on the objects and their interaction components.

Scripted Interactions

Object interactions are not always that complex. Sometimes their means of communication are relatively simple. For example, an engine can make an axis rotate by simply applying a torque to it.

Instead of implementing specific extensions for each of these simple interactions, a more scalable approach allows an interaction component to just "describe" in some scripting language how the participating objects communicate. We embedded an interpreter for the Python scripting language into the framework. Python scripts can be wrapped into a command description in the XML blueprint, as shown in figure 7.

The Python C API makes it possible not only to embed a python interpreter into an application, but also to extend the Python language with "extension modules" written in C, C++ and other languages. In order to allow Python to callback to the host application, in this case an application built on top of pLab, the host application was made an extension module itself. As the execution of the scripts is much slower than native C++ code, they are only useful in situations with simple interactions like the ones mentioned before.

The applications of the Python scripts are not limited to the implementation of interaction algorithms. They can be used to generate user interfaces, define animation sequences, perform operations on objects, etc.

5. SIMULATION EXAMPLE

This section elaborates on an example scene that combines all three interaction techniques mentioned in the previous section. A simplified schematic view on this scene is given in figure 8. The figure only shows the objects in the environment along with their interactions. Other components, like the simulation core itself, the collision library and the differential equation solver, are omitted but also form indispensable parts of the simulation.

The setup includes an engine that drives an axis. Mounted to this axis is a small gear that is coupled to another gear, which is twice as large and mounted to a second axis. The same axis contains a fan. In front of the fan are two balloons, a red and a green one. In this example, we will assume only the red balloon is close enough to the fan to get influenced by its airflow. The result of this particular simulation is that the fan starts rotating, which generates an airflow, and sets the red balloon on a collision course to the green balloon. The balloons collide and drift away. Figures 8, 9, 10 and 11 show the course of action.

Focusing on interaction components, the simulation starts when the script that handles the interaction between the gear and the axis applies a torque to the axis. This is a one-way interaction. The rotation of the axis is handed to the connected gear with a velocity constraint that ensures both objects have the same angular velocity. Constraints of the same type let the rotation propagate through the gears to the fan. Between the two gears however, the rotation direction has to be reversed and the speed has to be cut in half. This situation was also shown in figure 6.

Next, another script between the fan and the red balloon causes the red balloon to drift, according to the angular speed of the fan and their distance. When it collides with the green balloon, the collision library will detect and resolve the interference and apply the proper impulses to both objects.

6. FUTURE WORK

As indicated earlier, we need to work towards incorporating various simulation domains into one environment. The work described in this paper is a small step in that direction. Our approach was limited to the domain of rigid body dynamics, but it should be possible to extend this to the other domains like

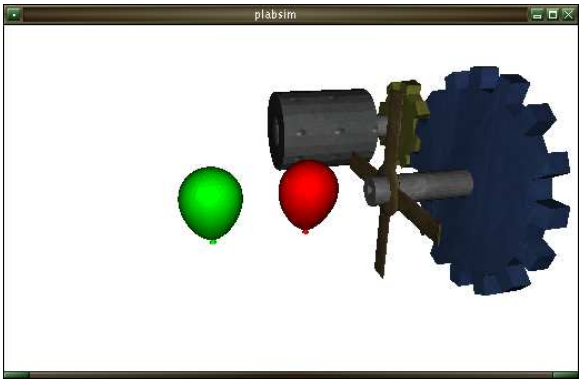


Figure 8. The initial setup of the scene.

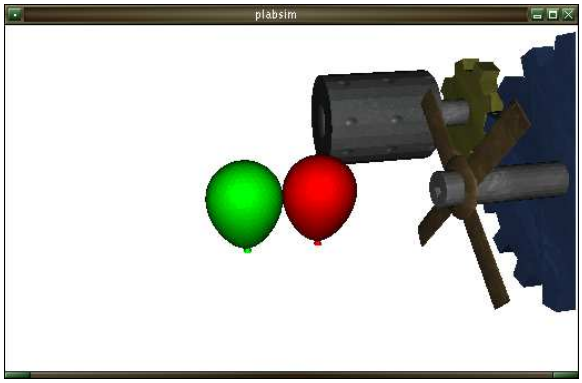


Figure 9. Collision time.

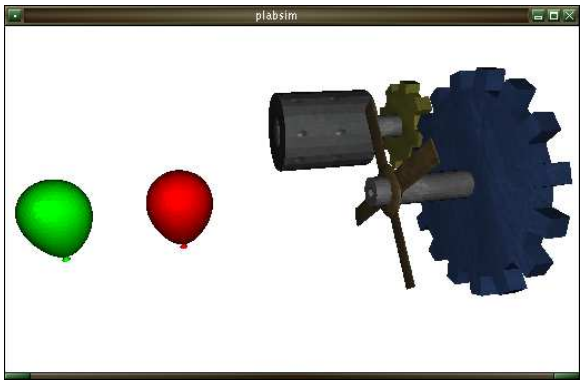
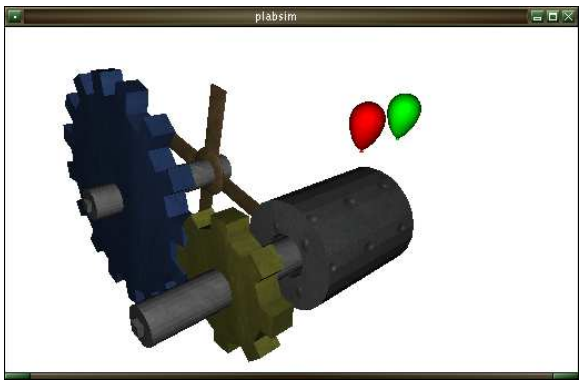


Figure 10. Both balloons drifting off.



Figuur 11. Scene viewed from behind.

cloth simulation and particle systems, and then further on to cross domain interactions.

The issue of exchanging interaction mechanisms at run time was not addressed in this paper. Doing so would greatly improve the chances of keeping complex virtual environments real time, without affecting the visual correctness of the simulation.

It would also be desirable if the user does not have to explicitly define the interaction mechanisms, but that they are detected automatically. For instance, one could specify just once what the behavior of a gear connected to an axis should be. This default behavior is then applied each time a user connects a gear to an axis. Also, if the objects would be aware of each other's proximity, they could be “snapped” together automatically by introducing a set of interaction mechanisms.

Finally, we only examined object-object interactions, but in virtual environments there are also user-object interactions. If the user is also defined as a participating object in the environment, he or she can be treated in the same way as the other objects and the techniques introduced in this paper also apply.

7. CONCLUSIONS

We extended the pLab framework with support for different types of interaction techniques, describing how objects in a virtual environment interact with each other.

To accomplish this, the interactions were generalized and viewed as black boxes, similar to the process of generalizing the objects in the scene themselves. Each interaction object received an interface with a simulation command that defines the interaction’s behavior during the simulation.

Finally, we demonstrated the results in a scene where three different types of interaction coexisted, namely collision detection and response, velocity constraints and interactions described by a scripting language.

8. REFERENCES

[Bar97a] Baraff, D., and Witkin. A. Partitioned dynamics. Technical Report CMU-RI-TR-97-33, Robotics Institute, Carnegie Mellon University, 1997.

[Bar97b] Barraf, D., and Witkin, A. Physically-based Modeling, Principles and Practice. Number 19 in Course Notes for SIGGRAPH’97. ACM, Los Angeles, CA, USA, August 3-8 1997.

[Ber99] van der Bergen, G. A fast and robust GJK implementation for collision detection of convex objects. Journal of Graphics Tools, 4(2):7-25, 1999.

[Cla01] Clark, J. Expat – XML parser toolkit. Software available at http://www.jclark.com/xml/expat.html, 2001.

[Got99] Gottschalk, S., and Lin, M. PQP – the proximity query package. Available at http://www.cs.unc.edu/geom/SSV/; 1999.

[Gre99] Gregory, A., and Lin, M.C. A framework for fast and accurate collision detection for haptic interaction. In Proceedings of Virtual Reality'99 Conference, pages 38-45, Houston, TE, USA, March 13-17 1999.

[Hud97] Hudson, T.C., and Cohen, J. V-COLLIDE: Accelerated collision detection for VRML. In Proceedings of VRML'97: Second Symposium on the Virtual Reality Modeling Language, Monterey, CA, February 24-26 1997.

[Kra90] Kramer, G.A. Solving geometric constraint systems. In Proceedings of the Eight National Conference on Artificial Intelligence, pages 708-714, Boston, MA, USA, July 29 – August 3 1990.

[Lae02] Van Laerhoven, T., and Van Reeth, F. The pLab project: An extensible architecture for physically based simulations. In Proceedings of Spring Conference on Computer Graphics, pages 129-135, Budmerice, SL, April 24-27 2002.

[Luy02] Luyten, K., and Van Laerhoven, T. Specifying user interfaces for runtime modal independent migration. In Proceedings of CADUI'2002 International Conference on Computer-Aided Design of User Interfaces, pages 238-294, Valenciennes, FR, May 15-17 2002.

[Mir96] Mirtich, B. Impulse-based Dynamic Simulation of Rigid Body Systems. PhD thesis, Berkeley University of California, 1996.

[Smi01] Smith, G., and Stuerzlinger, W., Integration of constraints into a VR environment. In Proceedings of Virtual Reality International Conference, Lava Virtual 2001, pages 103-110, Laval, FR, May 16-18 2001.

[Wit90] Witkin, A., and Gleicher, M. Interactive dynamics. Computer Graphics (1990 Symposium on Interactive 3D Graphics), 24(2):11-21; 1990.

[Woo99] Woo, M., and Neider, J. OpenGL Programming Guide. Addison-Wesley, 3 edition, 1999.

[W3c02] World Wide Web Consortium. Extensible markup language (XML). Web pages avalailable at http://www.w3.org/XML/, 1998-2002.

[Xgl02] XGL Working Group. XGL file format specification. Web pages available at http://www.xglspec.org/, 2001.