Interner Bericht

A 3D-Modeling- & Visualization-Toolkit for web-based Applications

Andreas Disch, Philip Jacob, Michael Münchhofen

295/98

Universität Kaiserslautern
Fachbereich Informatik
AG Computergraphik
Postfach 30 49
67653 Kaiserslautern

Fachbereich Informatik

Universität Kaiserslautern · Postfach 3049 · D-67653 Kaiserslautern

A 3D-Modeling- & Visualization-Toolkit for web-based Applications

Andreas Disch, Philip Jacob, Michael Münchhofen

Universität Kaiserslautern

295/98

Universität Kaiserslautern
Fachbereich Informatik
AG Computergraphik
Postfach 30 49
67653 Kaiserslautern

Jan. 1998

Herausgeber: AG Graphische Datenverarbeitung und Computergeometrie
Leiter: Professor Dr. H. Hagen

VIPER

A 3D-Modeling- & Visualization-Toolkit for web-based Applications

Cand. Dipl.-Inform. Andreas Disch (disch@informatik.uni-kl.de), Dr. rer. nat. Philip Jacob (jacob@informatik.uni-kl.de), Dipl.-Inform. Michael Münchhofen (mmuench@informatik.uni-kl.de).

Universität Kaiserslautern / AG Computergraphik

Report 295/98

http://viper.informatik.uni-kl.de/paper/97

Abstract

The World Wide Web is a medium through which a manufacturer may allow Internet visitors to customize or compose his products. Due to missing or rapidly changing standards these applications are often restricted to relatively simple CGI or JAVA based scripts. Usually, results like images or movies are stored in a database and are transferred on demand to the web-user. Viper (Visualisierung parametrisch editierbarer Raumkomponenten) is a Toolkit [VIP96] written in C++ and JAVA which provides 3D-modeling and visualization methods for developing complex web-based applications. The Toolkit has been designed to built a prototype, which can be used to construct and visualize prefabricated homes on the Internet. Alternative applications are outlined in this paper. Within Viper, all objects are stored in a scene graph (VSSG), which is the basic data structure of the Toolkit. To show the concept and structure of the Toolkit, functionality, and implementation of the prototype are described.

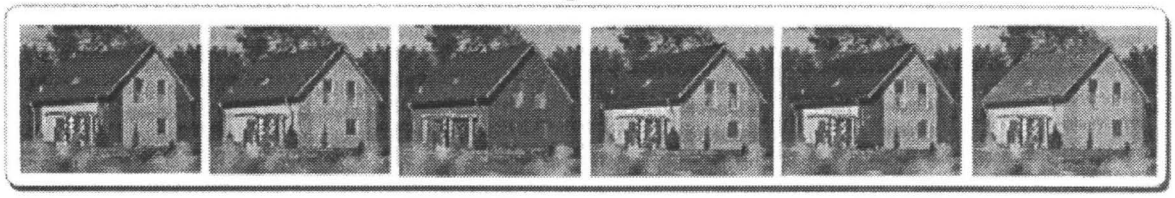
Keywords: JAVA, Cosmo Player, VRML, WWW, Modeling, Rendering, Visualization, Object-Oriented Design

1. Introduction

Using modern production techniques, many products can be individually customized and manufactured. The changing of parameters by customers results in a huge number of variants which cannot be represented with traditional techniques like print media. This is one of the main

problems of manufacturers of homes, where thousands of combinations in materials and outfit exist which cannot all be printed in catalogs. Kampa, one of the biggest manufacturers of prefabricated homes in Germany, is one of the partners of the Viper-Project. The following images show only a few of the possible 976 combinations, which exist for a single type of home.

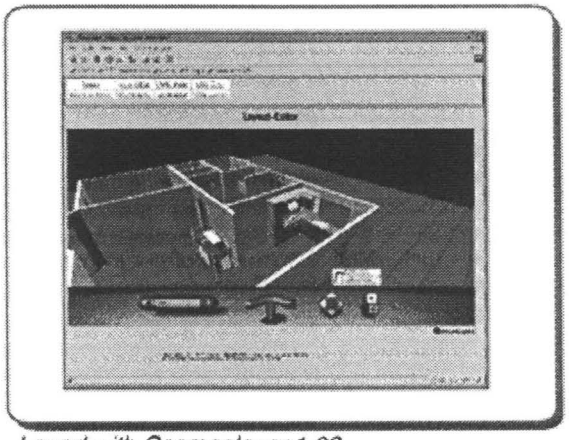
Figure 1



Kampa-Smart variations

Because of technical reasons (e.g. statics), some of these combinations are not valid or are simply not offered. During the planning of a home these constraints have to be surveyed by the system and eventually be corrected.
The modeling part of the Viper Toolkit on the server side was implemented in C++ to guarantee the performance needed for these operations. The average Internet connection has a bandwidth of less than 2 or 3 kbytes per second which is too small for an interactive construction process without using local routines. Therefore, the construction and modification of parameters is done by JAVA on the local PC.
Information is exchanged using a common data-structure, the VSSG (***Viper Symbolic Scene Graph).*** A JAVA-based editing tool (***LayEd***), which uses these structures to communicate with the kernel has been developed as a construction front end.

Figure 2



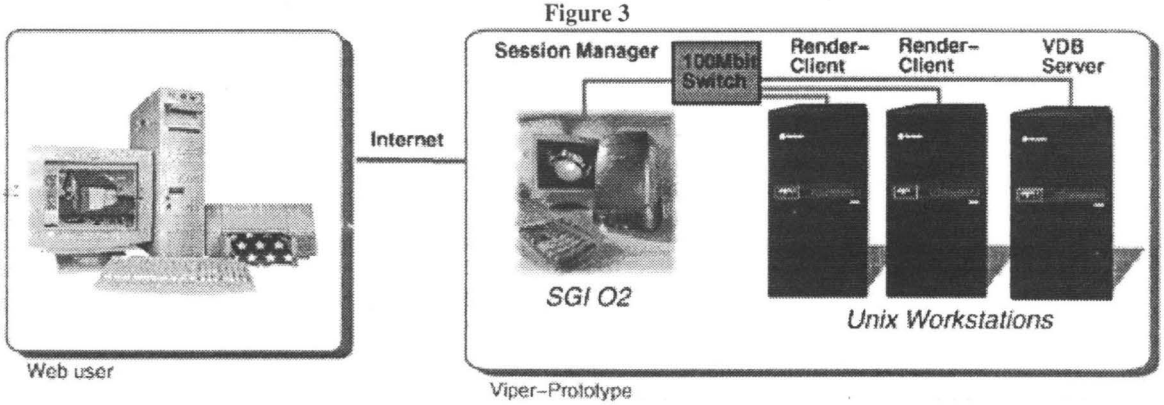
Layed with Cosmoplayer 1.02

In the following section we describe the prototype for planning and visualizing homes on the Internet to demonstrate functions and usage of the Toolkit. In addition the underlying data structures and implementation concepts are outlined.

2. The Prototype

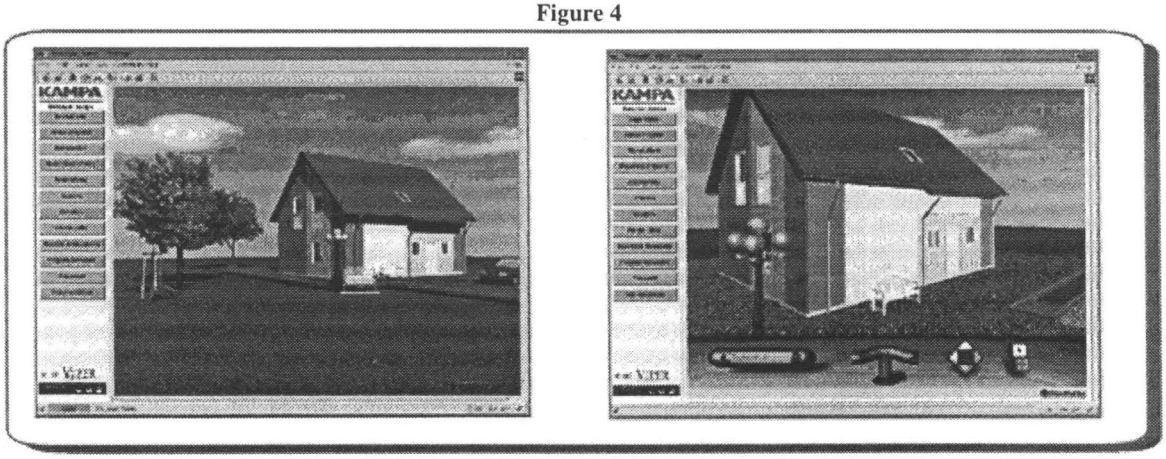
The prototype had to be independent from the operating system and hardware used by the customer. The system should allow the web-user to construct and render a home from his PC, without having installed any additional software beside an Internet browser. All functions of the prototype are provided by a session manager which is a web-interface running on an Apache web-server.
The prototype was setup on a cluster of Silicon Graphics O2 workstations, connected with a 100Mbit switch. After authorization the prototype can be accessed from any PC having a web-browser and a VRML-plugin (e.g. Cosmo Player).

Figure 3



Remote-rendering brings photo-realistic images and movies to the web-user without investing much time or resources on the client side. A distributed renderer generates the images. Results like MPEG-videos or single JPEG-snapshots are brought to the client by the World Wide Web. Alternatively, the Viper Toolkit creates a VRML-scene which can interactively be traversed.

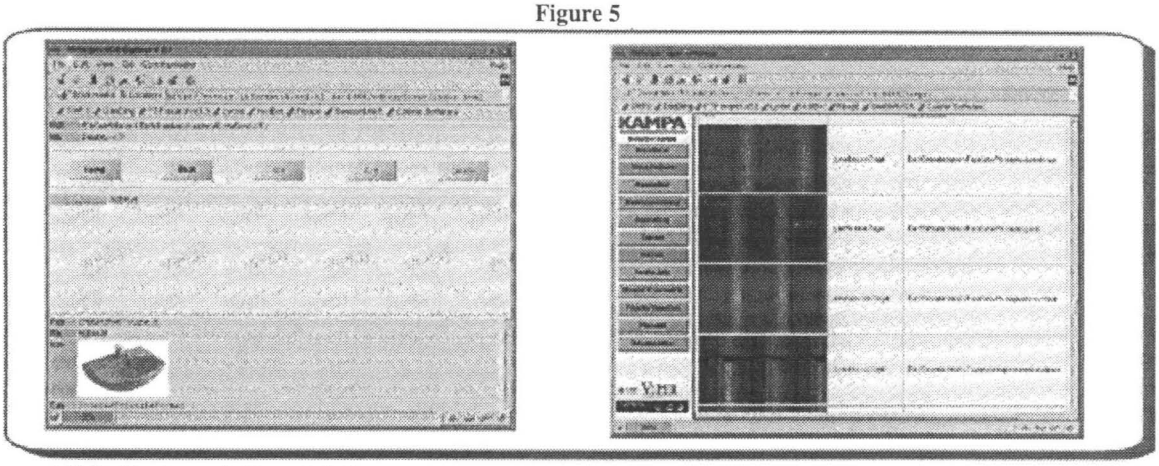
Figure 4



Rendered Image & VRML-scene

All objects like VSSG's, geometry, and materials are stored in a SQL-database which is administrated via WWW-interface (Viper-Database-Explorer). Lighting and cameras are directly modified by the user. To change materials and other attributes a parametric editor is provided.

Figure 5



VDB-Explorer & Material-Parametric

Constructions can be exchanged with CAAD-systems using the AP225 STEP-interface designed for architectural applications. After the construction is finished all information is transferred to the manufacturer and can there be used to initiate the production process.

3. The Viper Kernel and the Symbolic Scene Graph

3.1 An Overview

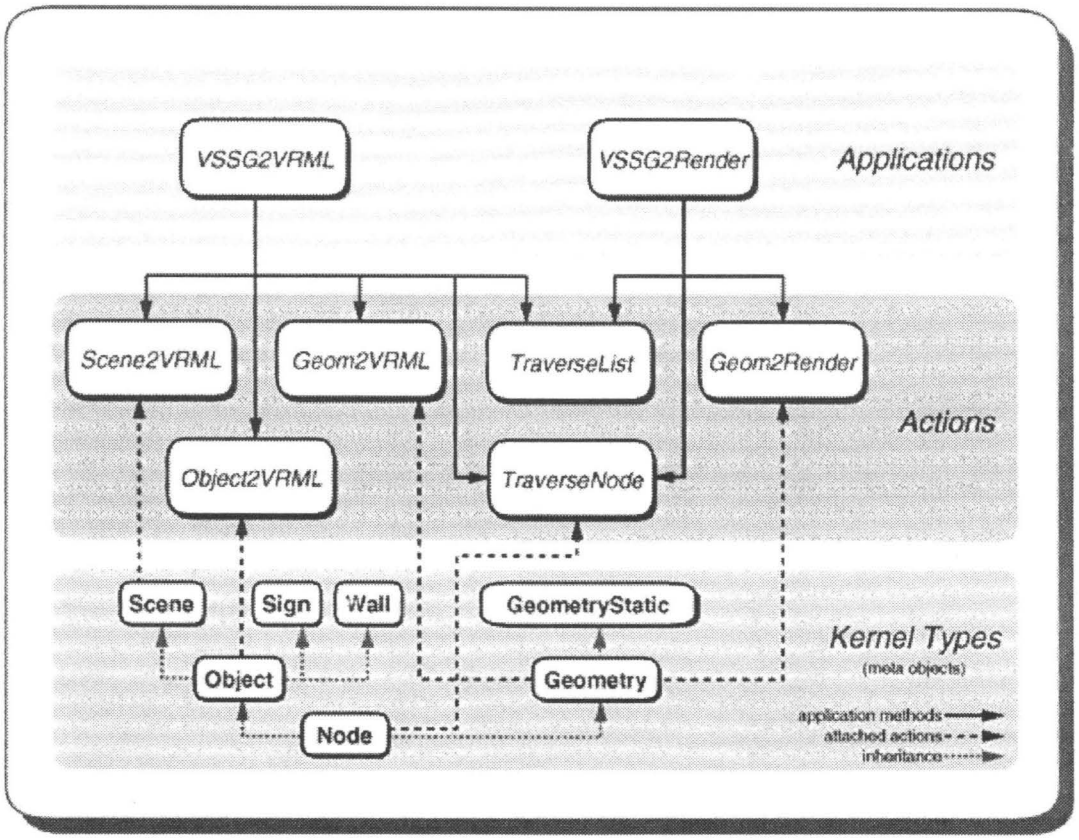
The purpose of the Viper Toolkit is to provide a framework for building complex, web-based graphics applications. To manipulate objects from different domains, the Viper Kernel manages all data in a unified, application-independent format. The object-oriented design of the toolkit allows high-level descriptions of application data because the basic idea is to describe objects and their relationships *symbolically*.
The common data structure maintained by the kernel is an abstract, dynamically defined description of an application's world: the Viper Symbolic Scene Graph (VSSG). The basic functionality of the kernel is implemented in a library, consisting of elementary applications written in C++.

3.2 Basic Concepts

While developing the Viper Kernel, emphasis has been placed on abstract processing of application data in a uniform style. Thus, all kinds of data are treated as *data objects* (so-called VSSG-Nodes), i.e. instances of generalized data types - *classes* - that may have attributes, which are either elementary or (references to) other data objects. Classes are organized in an inheritance hierarchy. Subclasses inherit the attributes from their parent classes and may extend the description with new attributes. Type-specific behavior is added through modular actions

(*methods*) which may be attached to data types individually for each application. A special dispatching mechanism is used to invoke these methods according to the types of a given object and application. This takes into account inheritance and results in a *multi-polymorphism*, comparable to the generic dispatching procedure of the Common Lisp Object System [CLOS].

Figure 6



Application Structure Example

To make the system as flexible as possible, the kernel defines only elementary data types and generic procedures, for example input and output routines allowing persistent storing. Additionally, each application gives the kernel a description of its own data types at run-time that contains the names of the classes, their base classes, and attribute declarations consisting of type, name, and default value. Within the kernel, each class declaration is dynamically assigned to a *meta object*, which represents that class and provides different features, e.g. dynamic creation of class instances, run-time type information or dispatching of multi-methods. Naturally, meta objects represent not only dynamic types but also static classes that are defined at compile-time. Therefore, the dynamic declaration of data types along with a mechanism for dispatching modular procedures equates logically with the static definition of classes and class hierarchies.

3.3 An Example

As mentioned earlier, there are two basic elements defining the structure of the VSSG:

VSSG-Attributes
are either elementary data types (numbers, strings, vectors, lists ...) or (symbolic references to) VSSG-Nodes.

VSSG-Nodes
consist of an exactly specified set of VSSG-Attributes, accessed by their unique name. Note that an attribute name
is not a property of the attribute itself, but rather a name for its *role* within the enclosing node.

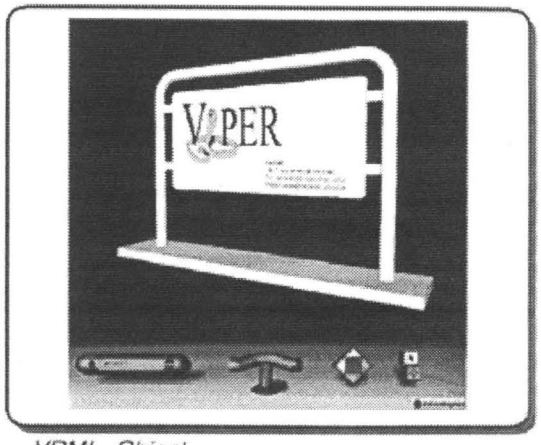
This approach allows hierarchical data structures to be built: the VSSG trees.
The following example of an application in an architectural environment should give an impression of modeling with the VSSG. A type declaration for modeling houses and environment may look like this:

```
Object : Node { Transform transform({}); }
# First, we define a new type for transformable objects.
# We derive it from 'Node', a predefined base type that
# contains a single attribute 'name' of type 'String'.
# The attribute 'transform' holds any transformations
# derived from 'Transform'; the default value is an
# empty node.

Transform : Node { }
# An abstract transformation type.

TransformSRT : Transform {
  Vec3d scale(1.0 1.0 1.0);
  Vec4d rotation(1.0 0.0 0.0 0.0);
  Vec3d translation(0.0 0.0 0.0);
}

# Standard transformation type to scale, rotate, and translate.
Scene : Object { List<Object objects({}); }
# The attribute 'objects' stores a list of children describing
# a scene. Members of the list must be of type 'Object' or derived
# from it.

Villa : Object { List<Floor floors({}); }
Floor : Object { List<Wall walls({}); }
Wall : Object { Geometry geometry({}); }
Sign : Object { Geometry geometry({}); }
Geometry : Node { Material material({}); }
GeometryStatic : Geometry { String filename(""); }
# We want to read static geometries from a file.
...
```

Once that we have defined these types, an application like a converter to VRML is ready to process a VSSG based on the above declaration.

```
# VSSG V3.0 ascii
Scene {
  name "simple VSSG"
  objects {
    Sign {
      name "ViperSign"
      transform TransformSRT {
        scale 0.7 0.7 0.7
        translation -5 -0.15 3
      }
      geometry GeometryStatic {
        filename "VDB:H40P44/ITF/Sign.itf"
      }
    }
  }
}
```

Figure 7



VRML-Object

Using the Viper Toolkit the simple VSSG can for example be rendered or converted to VRML. We include a complex VSSG of the prefabricated home shown in figure 4 and the corresponding VRML-Scene.

4 Java Based Modeling And Visualization Toolkit (JaMVis)

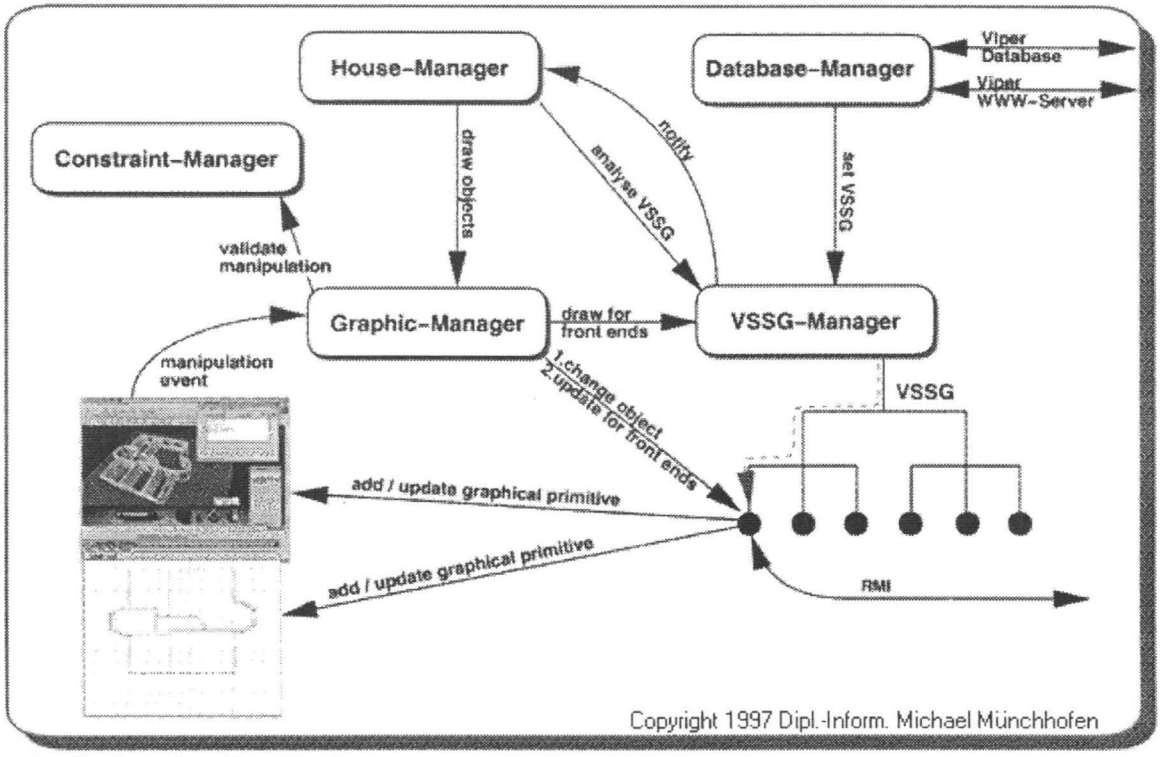
4.1 An Overview

In the VIPER Project we need a toolkit which is based on internet technologies for the visualization and modeling over the net. The CGI and database interfaces were not flexible enough and did not provide any real-time interaction. On that account we developed the JaMVis Toolkit. The toolkit consists of a variety of modules where each solves only a small task. A task can be generic like a database access or the visualization of three dimensional objects. Specialized tasks have the knowledge about the structure how an application should work. An example for that is a task which manages the modeling of a house. We use JAVA [JAVA] as the programing language and so the applications can be used within a web-browser or as a standalone application.
As described in the section three we use the Viper Symbolic Scene Graph as the fundamental data structure. This allows us to model our world without assigning a semantic to the data. Time critical tasks have a native interface to other high level programing languages e.g. for the efficient use of graphics hardware. Based on the JamVis Toolkit we developed an editor for modeling the floors of a house. The editor consist of several JAVA-Beans [BEAN], which communicate via the event model.

4.2 The Modules

LayEd is an editor for designing the floor plans. It is a client-server application and consists of 100% pure JAVA code on the client side. In most cases the client side computer is not a high end computer. So we had to shift the cost expensive algorithms to a high end Unix machine. This machine is responsible for the data conversion and processing. We work with data sets which consists of several hundreds of kilobytes. On the client side we need reduced information. therefore, we convert the data on the server into the desired format and send it to the client. The communication between the modules is given in figure 8.

Figure 8

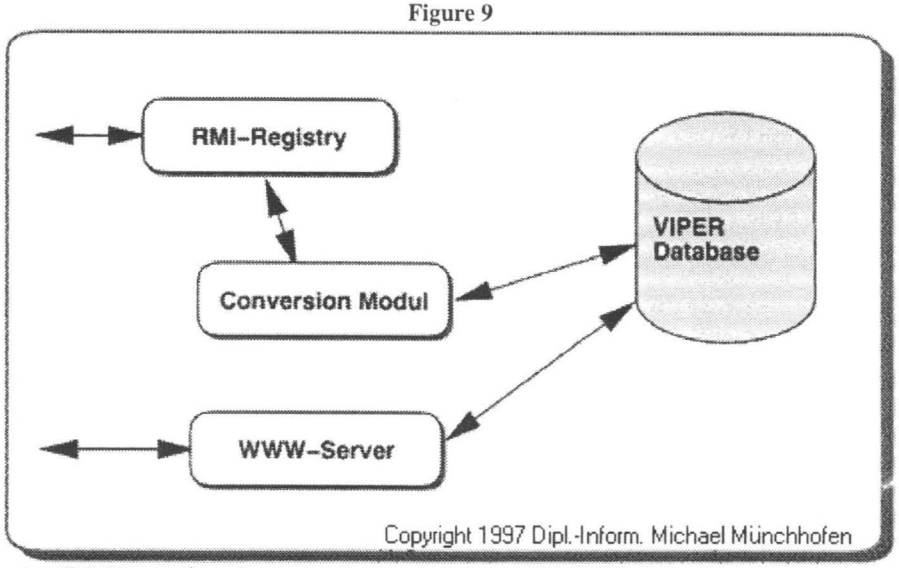


LayEd: Simplified Client Structure

The Database-Manager is responsible for receiving the data from the server. The data (ASCII format) is transferred to the VSSG-Manager. This module parses the data and builds the data tree. After that, it notifies the House-Manager. In the editor we visualize each floor separately. Therefore, this module analyzes the VSSG to receive the data for each floor. After building the control structure the House-Manager sends a message for each object that has to be drawn to the Graphic-Manager. It starts the generation of the graphics objects for each front end. The front ends (2D/3D-Viewer/-Editor) must be registered in the Graphic-Manager to receive the graphical objects. Therefore, the Graphic-Manager starts an action for each front end.
In the editing process a user sends a message about the manipulation of the data to the Graphic-Manager. The manipulation event is generated by the front end which evaluates the user interaction. The manipulation is validated by the Constraint-Manager.
There are several dependencies in the house, e.g. positioning of some objects (like a washing-

basin) needs an installation wall. So if there is a violation in the rule set, the user will be informed by the Constraint-Manager or the system tries to adapt the floor plan, e.g. changing the wall. After validation the destination VSSG objects are updated by the Graphic-Manager.

Figure 9



LayEd: Server Structure

The server depends on the Remote Method Invocation Interface [RMI] provided by JAVA. Due to the generation of the graphical objects the client sends several requests to the server. The server may starts C++ programs for conversion into the desired format on several hosts. Because of speed we use the system command of Java. We made the experience that accessing the socket interface (via RMI or CORBA) is slower than the usage of the remote shell system command from the Unix operating system.

4.3 Implementation

The central bean is our VSSG-Manager. It is responsible for the actual floor plan. In addition to the C++ implementation of the VSSG we needed the same structure in JAVA. We use the VSSG type definition to generate JAVA classes which are combined in a package for each destination application. The data classes include fundamental methods for input and output operations and for accessing the attributes. As in C++ the classes do not contain any application specific methods. For each application we need a set of several actions for the classes, e. g. transformations for the VSSG nodes. The actions are combined to a package which we call a VSSG-Application. The inheritance chain from the data classes is used to associate the actions. To initiate an action in the VSSG structure the VSSG-Manager provides an event interface.
Graphical beans are managed by the Graphic-Manager. At creation time each graphical front end must register at the manager for receiving his corresponding graphical objects. They are administered in a list of all active editors and viewers. With that the Graphic-Manager can initiate the actions for generating the graphical data structure (e.g. VRML, Open Inventor, Open GL). The generated structures are transferred by an event to the associated front end.
Manipulation of objects is done by an editor. It sends a request for manipulation to the Graphic-Manager. After validating the request, it is forwarded to the data object. The object modifies the data and updates the graphical primitives in the front ends. The Constraint-Manager validates the

requests and is responsible for the consistency of the floor plan. If a manipulation violates a rule and the manager can repair the inconsistency, then it stops the manipulation event and takes over the control for the action generation. Now the Constraint-Manager sends new requests to modify the floor plan in a consistent way.
LayEd is build with this modules using the Cosmo Player 2.0 beta from Silicon Graphics via External Authoring Interface. The user can load a floor plan from a database and modifies the floors. He can also endow the floor with furniture and other fixings. The new plan can be visualized with the protoype.

5. Conclusion & Future Work

Even though the system has been designed to be platform independent we had problems using VRML and JAVA especially with 'older' versions of web-browsers and VRML-plugins. During the development of the prototype we found several alternative operational areas for the Viper Toolkit, like manufacturing of furniture and interior design. Especially for these applications it would be very helpful for the user to have a global illumination model integrated in the renderer to include indirect illumination in the images or to calculate radiosities for the VRML-scenes.

6. References

[APACHE] Apache HTTP Server Project http://www.apache.org/
[CLOS] Bobrow, Daniel G., Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, David A. Moon. *Common Lisp Object System Specification*, X3J13 Document 88-002R, June 1988.
[EAI] The external authoring interface http://vrml.sgi.com/developer/eai/index.html
[VRML2.0] The Virtual Reality Modeling Language http://vrml.sgi.com/basics/index.html
[JAVA] The JAVA™ programming language http://www.sun.com/java/
[Open GL] Open GL http://www.sgi.com/Technology/OpenGL/
[Open GL Opt] Open GL Optimizer™ http://www.sgi.com/Technology/OpenGL/optimizer/
[OpInv] Open Inventor http://www.sgi.com/Technology/Inventor.html
[BEAN] JAVA-Beans http://www.javasoft.com/beans/index.html
[VIP96] Viper Report 1995/96
[RMI] The Remote Method Invocation http://java.sun.com/products/jdk/rmi/index.html
[CORBA] The Common Object Request Broker Architekture
http://www.omg.org/news/begin.htm