Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

IVORY - An Object-Oriented Framework for Physics-Based Information Visualization in Java.

T. C. Sprenger, M. H. Gross, D. Bielser, T. Strasser

Proceedings of IEEE Information Visualization '98
Research Triangle Park, NC, USA
October 19-20, 1998, pp. 79-86, 1998

We present IVORY, a newly developed, platform-independent framework for physics-based visualization. IVORY is especially designed for information visualization applications and multidimensional graph layout. It is fully implemented in Java 1.1 and its architecture features client-server setup, which allows to run the visualization even on thin clients. In addition, VRML 2.0 exports can be viewed by any VRML plugged-in WWW-browser. Individual visual metaphors are invoked into IVORY via an advanced plug-in mechanism, where plug-ins can be implemented by any experienced user. The configuration of IVORY is accomplished using a script language, called IVML. Some interactive visualization examples, such as the integration of an haptic interface illustrate the performance and versatility of our system. Our current implementation supports NT 4.0.

IVORY - An Object-Oriented Framework for Physics-Based Information Visualization in Java

T. C. Sprenger, M. H. Gross, D. Bielser, T. Strasser

Department of Computer Science, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland

ABSTRACT

We present IVORY, a newly developed, platform-independent framework for physics-based visualization. IVORY is especially designed for information visualization applications and multidimensional graph layout. It is fully implemented in Java 1.1 and its architecture features client-server setup, which allows to run the visualization even on thin clients. In addition, VRML 2.0 exports can be viewed by any VRML plugged-in WWW-browser. Individual visual metaphors are invoked into IVORY via an advanced plug-in mechanism, where plug-ins can be implemented by any experienced user. The configuration of IVORY is accomplished using a script language, called IVML. Some interactive visualization examples, such as the integration of an haptic interface illustrate the performance and versatility of our system. Our current implementation supports NT 4.0.[1]

keywords: three-dimensional information visualization, physics-based graph layout, object-oriented visualization toolkit, multidimensional information modeling, time varying data.

1 INTRODUCTION

Visual communication had always been of fundamental importance to mediate information and to understand complex relationships. With the advent of the computer, scientific visualization was born as a discipline [1] and conquered many science and engineering applications. However, whereas in the past, research was mostly focussed on the visualization of spatial data sets and metric spaces, the design of new visual metaphors for abstract, complex information spaces has recently emerged as a challenging research topic. Applications are manifold and range from web/network visualization and document retrieval, via software engineering to risk and portfolio management in the financial services [11].

Due to the tremendous importance of visualization methods various systems and toolkits had successfully been designed in the past, part of which are available as commercial products. One of the pioneers is AVS/Exress [2] that uses a data flow paradigm and allows the user to compose a visualization application interactively by definition of data flow paths between individual modules. Similar paradigms have been implemented in the IRIS Explorer [3] or in IBM's DataExplorer [4]. Another elegant visualization library is provided by General Electric's VTK [5] and can be customized in TCL/TK. However, most of the general purpose toolkits and libraries target at classical scientific visualization of spatial data.

For information visualization and visual data mining sophisticated algorithms and metaphors had been devised in recent years to visually inspect abstract and multidimensional information spaces. Cone trees [6] and their hyperbolic projections [7] are only one prominent example. Physics-based graph layout is another important paradigm and has been successfully exploited in various applications [8], [9], [10]. Good surveys of contemporary information visualization methods can be found in [11] or [12]. However, generic toolkits and systems are rare or mostly focussed, such as statistics packages, like Bell Lab's XGobi [13], IVEE [14] or the hierarchical algorithms in SGI's SiteManager. Conversely, visual data mining tools, like SGI's MineSet [15] often comprise visualization functionality, however, have limited flexibility regarding the integration new metaphors.

[1] IVORY will be made available in short.

2 OUR APPROACH

Our toolkit IVORY has been developed as an open and flexible system for information visualization to fill the gap between general purpose visualization tools and application specific systems. Although it is primarily designed for physics-based approaches to multidimensional information spaces and for the visual analysis of large financial data volumes, the underlying design principles make it a versatile framework for the investigation and application of new visual metaphors. Some algorithmic details of the visualization paradigm currently used in IVORY have been presented by the authors in [9] and [16].

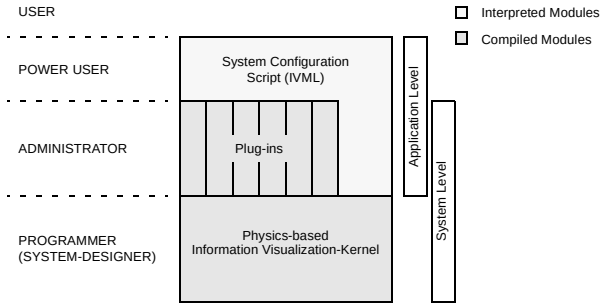


Figure 1: User abstraction model and system architecture

Our essential design and engineering goals for IVORY can be summarized as follows:

- *Client-Server setup*: This allows the separation of any visualization mapping engine from the client and the user interface.
- *Java programming language:* Despite of all hype, Java features object-oriented programing paradigms and stands for platform independence.
- *Clustering and hierarchies:* In order to manage complexity of information objects IVORY supports specifically multiresolution visualization methods, such as hierarchies and clustering [9].
- *4 layered abstraction model for users*: In order to offer an appropriate interface for different types of users, IVORY can be configured at 4 different levels of abstraction, depicted in Figure 1. The *standard user*, such as a financial analyst, will apply preconfigured instances of IVORY in his everyday work. The more advanced *power user* can customize existing configurations using the script language IVML. On an *admin-*

istrators level Java plug-ins can be programmed to implement individual visual metaphors and layout algorithms. Finally, the *systems designer* can modify and extend the IVORY kernel to add new classes for solvers, particle engines or other kernel methods.

The remainder of the paper is organized as follows: In section 3 we elaborate on general design and architectural issues underlying our system. Section 4 addresses the plug-in mechanism, which allows to implement new visualization paradigms. Next, we discuss the scripting language, where, however, for brevity, the complete EBNF is omitted. Finally, an example using a force-feedback interaction model illustrates the flexibility and versatility of our approach.

3 ARCHITECTURAL ISSUES

To implement the introduced system architecture shown in Figure 1 we choose an object-oriented framework. Our basic design goals include a fast, compact kernel and a powerful interface which can be flexibly expanded for a wide range of user-specific problems.

3.1 Framework Concepts

When designing our toolkit, we focused specifically on physics-based information visualization, since it provides a promising and intuitive paradigm enabling the user to map abstract multidimensional information spaces onto appropriate subspaces, that can be visualized in 3D or 2D.

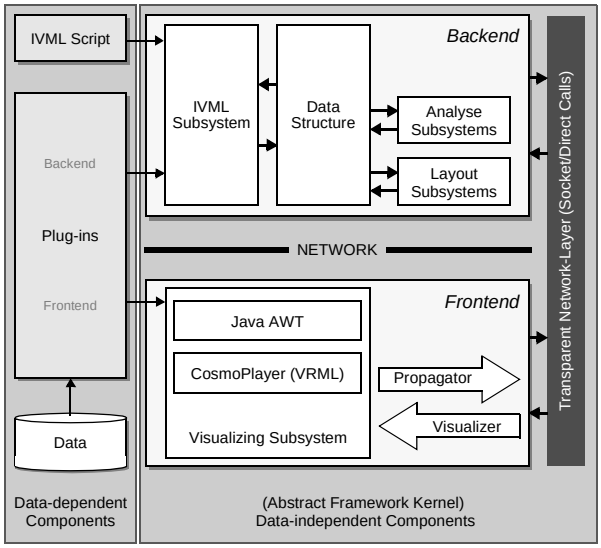


Figure 2: Schematic overview of the system-components in IVORY

Double Separation

As illustrated by the horizontal separation in Figure 2 we employed a frontend/backend-concept, where the backend is responsible for efficient number crunching and the frontend handles the graphical user interface and user interactions. In addition, we distinguish between data-dependent and data-independent components (vertical separation in Figure 2). For this reason we distill all data-dependent components into so-called plug-ins. The script language IVML handles configurations of individual plug-ins. That is, in order to visualize a new data-type, the user has essentially to write an appropriate plug-in. The remainder of the system is not affected at all. For a more detailed discussion of the plug-in concept we refer to Section 4.

System Configuration using IVML

The configuration and parametrization of the kernel and the plug-ins to be loaded, has both to be comfortable for the users and to exhibit small turn-around times. Therefore, the framework is equipped with a high-level script language, called IVML (Information Visualization Modeling Language). In contrast to all other system components which are build by using a compiled language providing a maximum of speed, the configuration scripts are handled by an interpreter. Since they execute only once at start-up time we balance time against flexibility.

All in all, the fundamental question of compiled or interpreted components is solved by a compromise: On the one hand the interpreted approach is used wherever the execution repetition is very low and therefore the demand for speed is of secondary interest. On the other hand compiled code is used for all low-level and often-run-through program segments.

Some Arguments pro Java

Two important aspect have to bear in mind to give a framework the chance for general acceptance: First the system should base on already established standards. Second, the implementation should be as independent of operating systems as possible.

In this context, a new trend in portability emerges with Java [17] and its *"write once, run everywhere"* philosophy. In this way applications run (almost) on any operating system supporting Java. We believe that the leak of performance compared to C++ will get obsolete over time. Here, the performance improvements made with every new Java release and the availability of native code Java compilers [18], [19] are good evidences.

In addition, besides all the euphoria, Java is widely accepted as a highly portable quasi-standard in the computer community and is available on many operating systems. Furthermore Java is strongly object-oriented, which is critical for a well structured implementation of a complex framework.

For 3D graphic output we selected the system independent modeling language VRML 2.0 [20] to provide a high-level description of the scene. Specifically, there are many viewers and APIs available supporting VRML 2.0. Each of them handles basic object management such as adding, removing and picking of an object.

3.2 Information Flow

In our terminology an information flow affects the visual representation of our input data. Consequently, one stream of information flow is the data itself. It starts with the data acquisition and ends up in the visual representation. If we connect, for instance, real-time data feeds, such as a Reuters ticker, to the system the visualization will be continuously driven by the data. A second important flow of information is defined by the user interactions. The information contained in this flow range from mouse clicks to cursor updates of a connected haptical device.

Data Acquisition

The data acquisition is the starting point of every visualization or visual mining procedure and supplies the system with raw material. In the simplest case the data can be read from an ASCII-file. Since the acquisition process obviously belongs to the data-dependent components and is therefore settled in the data plug-in components, IVORY does not impose any further restrictions on data access. Each instance of a data plug-in is responsible for its own data acquisition, which is thus fully transparent to the rest of the system. This mechanism enables us to tap any data-source by a specifically designed method and conversely to access one data-type through different methods without changing the rest of the

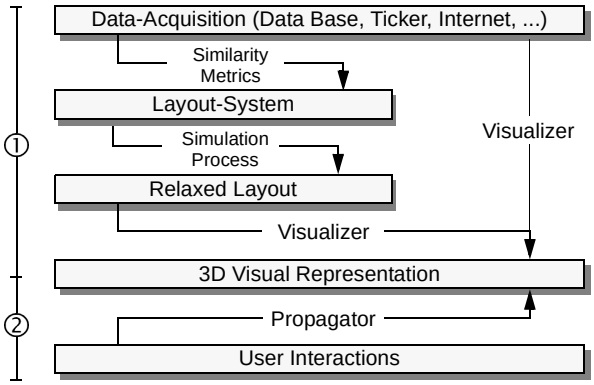


Figure 3: IVORY's information flow pipeline. 1) Data driven information flow. 2) User interaction (feed-back) information flow

system. Typical data-sources encompass conventional data bases (SQL, DB2, DBase, ...), search-engines in the internet, the World Wide Web (WWW) or real-time data tickers.

Configuration of the Layout System

In many graph-based visualization methods, data units, as represented in the system by instances of plug-ins, are considered as entities in a physical world and relations between them can be mapped onto interacting forces. However, the configuration of an individual layout algorithm takes a direct influence on the resulting object arrangement in 3D space [8], [10] and [16].

In IVORY, the configuration of the interacting forces is done by a second type of plug-in, the connection plug-ins. They provide two adjustable parameters, rest length l_0 and the spring stiffness k. Their values are defined through the metric method of the connection plug-in. This method calculates a scalar value depending on similarity s_{ij} of the two connected data plug-ins d_i and d_j. The definition of expressive metric functions is still an ongoing research issue in information retrieval and can therefore be individually adjusted.

In the current setting we propose four different mapping models summarized as follows:

- Similarity s_{ij} mapped onto the rest length l_0 only:
  $k = const$ (Chose $const$ sufficiently large)
  $l_0 = inv(s_{ij})$ (Rest length inverse proportional to stiffness)

- Similarity s_{ij} mapped onto the spring stiffness k only:
  $k = s_{ij}$ (Stiffness proportional to rest length)
  $l_0 = const$.

- Similarity s_{ij} mapped onto both parameters l_0 and k:
  $k = s_{ij}$ and $l_0 = inv(s_{ij})$.

- Similarity s_{ij} mapped onto the rest length l_0 and the reliability r of the value of s_{ij} mapped on the spring stiffness k:
  $k = r(s_{ij})$ and $l_0 = inv(s_{ij})$.

All methods mentioned above are aiming at a mapping of object similarities onto the parameters of the physics-based metaphor. That means, similar data objects will be tighten together by a large k and/or a small l_0. The last version especially considers the reliability of individual similarity values and is used for the visualization of uncertain or unreliable data.

Relaxed Layouts

Self-organizing layouts are results of a simulation process relaxing the configured system into an energy minimum. This process eventually performs the dimensionality reduction and discovers knowledge in data or confirms existing hypotheses about the data. Specifically, in explorative data analysis, clustering and aggregation of data objects are extremely important methods. The layout algorithms used perform implicit cluster-based object arrangements, a more detailed discussion of which is given in [9].

3D Representation

The 3D representation is the essential step in the data information flow. In order to assign color, material, texture, geometry or other features to the data objects or aggregations IVORY provides the plug-in mechanism, where a data-specific property mapping could be defined for each data-type. The mapping from data properties to visual attributes is done by the visualizer task. In addition, IVORY provides a powerful set of user interactions described in the following subsection.

User interactions

The user interaction closes the feed-back loop by which the user can optimize the visual data analysis process. Besides generic interaction methods, such as free or constraint navigation in the 3D-representation or object picking, IVORY supports specifically a so-called "*drill-through*" mechanism. That is, when picking an object we can invoke any data-dependent operation on it, such as requesting a 2D-visualization of the underlying data or direct data editing. This is accomplished by implementing the appropriate method in the objects plug-in. In this way, for each data-type there are various data-specific interaction methods provided for the user. Generally, IVORY advocates the same intuitive interaction methods many users are used to from modern desktop windowing systems.

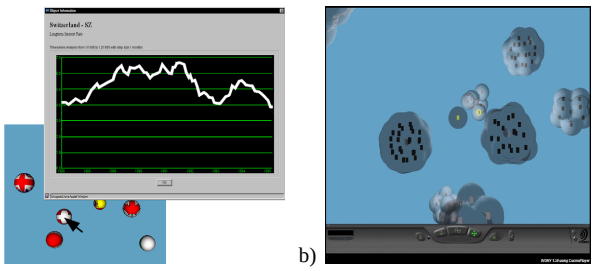


Figure 4: Results of possible user interactions. a) "Drill-through" the data provided by a currency plug-in. b) 3D Arrangement of a HTML-page domain (similarity criterion is the URL) after invoking the Blob-clustering algorithm from [9]. (See also Color Plates: Figure CP-1)

Another class of user interactions are the computer aided analysis methods. By the use of *view filtering*, for instance, the user can select interesting subsets of objects. Conversely, clustering is used as a method to handle complex visualizations, where groups of objects are condensed to one meta-object, a *cluster*. Applying this process recursively we can build hierarchical object arrangements. This enhances the overview and simultaneously reduces the computational efforts. A third class of methods are *pathfinders*. They are specifically designed for graph-based visualizations and enable to identify paths between individual nodes in a graph [21].

3.3 Framework Kernel

The abstract framework kernel of IVORY contains generic implementations of all common components underlying to our physics-based visualization paradigms and belongs to the set of data-independent components of the framework. Thus, kernel methods are highly reusable and shorten the design cycles of novel visual metaphors. Unlike most visualization systems, where the frontend/backend-separation is introduced to detach system-dependent from system-independent code segments, we employed this notion primarily to run the system in client-server setup over a network.

Frontend

The frontend is designed to run in a Java enabled WWW-browser. It consists of three parts: The visualization subsystem, which is responsible for all visual system outputs, as well as for the handling of user inputs. The 2D graphical user interface (GUI),

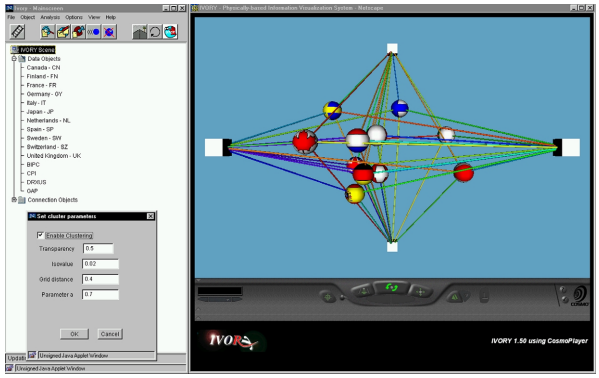


Figure 5: Screenshot of the IVORY frontend running on a Windows NT 4.0 machine under Netscape 4.04 with AWT 1.1 support. (See also Color Plates: Figure CP-2)

shown on the left side of Figure 5, covers all standard I/O tasks with appropriate menus and dialog boxes. Per default, an outline of all loaded objects (plug-ins) is shown as a collapsible tree structure. The 3D viewer is presented on the right side of Figure 5. Note that the visualized objects do not belong to the frontend. They manage the visualization of the calculated object arrangements and basic navigation functions. In order to decouple the frontend from the viewer we defined a generic 3D viewer interface. Thus, only the defined interface has to be re-implemented when changing the VRML viewer.

In addition we introduce two messenger components. The propagator is responsible to inform the backend, if a user interaction has invalidated the integrity of the frontend and the backend data-structures. E.g. whenever the user manipulates the layout parameters, the backend has to recalculate the corresponding object arrangement and synchronize itself with the frontend. The visualizer performs in a similar manner, but in the opposite direction from the backend data-structure to the frontend visualization.

Backend

The backend supports two different execution modes: Either it is directly attached to the frontend and runs in the same address space or it runs as a separate server-application on a different machine. In the second case the frontend and backend communicate over a Java socket connection.

One of the main design goals for the backend was a *responsive care data-structure*, where all the instantiated objects (plug-ins) are stored. It is strictly optimized for fast object insertion, deletion and look up. The structure is initialized by the IVML subsystem, which is responsible for dynamic object instantiating according to the parsed configuration script. Hence, this subsystem must be able to load and link unknown plug-ins at runtime.

As indicated by Figure 2, we distinguish between two different subsystem components accessing the core data-structure directly. The layout subsystem builds up a mass-spring-network based on the objects stored in the core structure. It also contains state-of-the-art differential-equation-solvers to simulate networks relaxation over time. This includes specifically gradient (Euler, Runge-Kutta) or stochastic (annealing) based-methods, where, in practice, gradient algorithms are much more suitable to treat time-varying data. The second subsystem type is provided by the analysis component comprising selection filters, pathfinders and clustering algorithms. The subsystem interfaces allow users at a *systems designer* level to easily extended kernel algorithms.

Transparent Network-Layer

All communication between frontend and backend is streamed over the transparent network-layer. Hence, the communication is generally transparent to all system components from above and guarantees full independence of the IVORY execution mode (stand-alone or client/server).

4 PLUG-IN MODEL

While defining the abstraction level of IVORY's plug-ins, we focussed on productivity and ease of use. Since Plug-ins only contain functions which implement a specific metaphor or rule the plug-in programmer can concentrate his efforts on the data-specific issues. Recalling the information flow pipeline shown in Figure 3, a plug-in contains, for instance, the similarity metrics and the property- and behavior-mappers driving the visualizers.

For reasons of simplicity, plug-ins contain both frontend- and backend-components. Conceptional, the separation is reflected by two groups of methods. Unlike AVS 5 [2], where separate computation and description modules exist, an IVORY plug-in is always viewed as one entity even though individual classes belong either to the client or to the server.

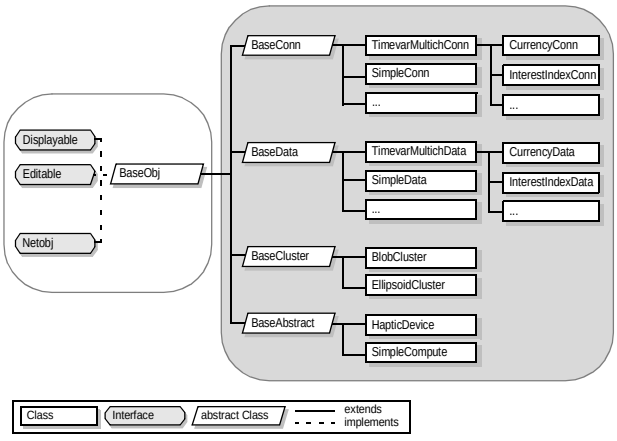


Figure 6: IVORY's plug-in hierarchy: The base classes.

The design of new plug-ins takes advantage of the object-oriented approach of the framework, where we make extensive use of object inheritance. All plug-ins are derived from a small set of base classes and are thus organized in a hierarchical structure, illustrated in Figure 6. The most important classes are discussed in the following subsections.

4.1 Base Objects

The base object class (`BaseObj`) builds the root of all plug-ins. Practically it's the only object class known in the abstract information visualization kernel of the system. It essentially determines the set of methods, the plug-ins could be accessed through.

A selection of the most important (abstract) methods are given below:

Frontend Methods

- **`getIcon`** (abstract)
  Optionally returns a reference to an icon resource. If available, it will be displayed in the 2D outline window.
- **`getAppearance`** (abstract)
  Returns the description of the representation of the object in the 3D viewer and is described in native VRML 2.0 [20].
- **`getDisplayComponent`** (implemented)
  This method provides the dialog component, which displays the information contained in the corresponding object. This includes administrative information (id, position, flags, ...) as well as stored user data.
- **`getEditComponent`** (abstract)
  If the object features editing, the method provides an editor component for the above information. This enables specifically data editing at runtime.
- **`getVisPar`** (implemented)
  Returns the previously calculated visual parameters by calling the method `calcVisPar`.

Backend Methods

- **`calcVisPar`** (abstract)
  Calculates the visual parameters (position, scale, orientation and color) of an object depending on the underlying data. This way, specific data-properties can be mapped onto visual attributes.

4.2 Data Objects

Data objects are directly derived from the base object class and serve as an access interface to the explored data. The data management is individually solved by the current implementation of a data object. For each instance of a data object a corresponding particle-mass is automatically created in the layout-subsystem. The setup of particle parameters is handled by the additional backend method explained below.

Due to its paramount importance for many applications we support time-variant data and multiple data channels per data object in our implementation. A good example is the analysis of a set of critical interest rates of different countries over the last years. In this case, a data object is allocated for each country. In each data object one channel is opened for each data feed [9].

Additional Backend Methods

- **`calcParticle`** (abstract)
  In this method non-visual parameters of the particle attached to the data object could be parametrized. In this way the object behavior during the layout process can be defined.

4.3 Connection Objects

Connection objects are of the same inheritance level as the data objects. This object type serves as a binding object, which represents the relation of two data objects and is of fundamental relevance for the resulting object layout.

For each connection object instance a corresponding spring is created in the layout-subsystem. The physical parameters of the spring are defined via the additional backend method described below.

Additional Backend Methods

- **`calcConnection`** (abstract)
  In this method non-visual parameters of the spring corresponding to the connection object can be parametrized.

4.4 Cluster Objects

Another object type of the first derivation level are the cluster objects. They enable a hierarchical organization of the visualization and can be looked at as data object containers.

Note, that they are created by the corresponding clustering algorithm located in the analysis subsystem. For each identified cluster a new instance is allocated. The appearance of this instance is a result of the cluster analysis calculations. In Figure 7 the resulting appearances of two different clustering methods are illustrated.

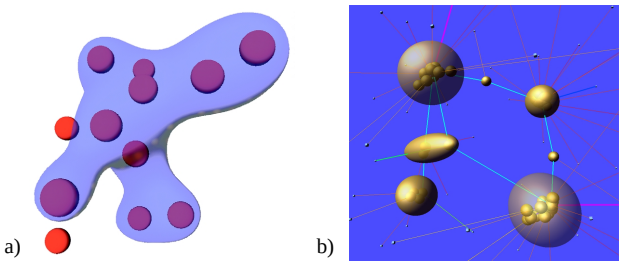


Figure 7: Two implemented types of clustering. a) Blob Cluster and b) Ellipsoid Cluster. (See also Color Plates: Figure CP-3)

4.5 Abstract System Object

Our experience has shown that generic objects without a visual appearance are very helpful for efficient solutions. Thus, we introduce so-called abstract system objects. Examples are parameterized global functions, such as data or currency converters.

Another area of application is the attachment of additional I/O-devices. For example, our physics-based system is predestinated for the use of force-feedback devices. In the example presented in Section 7, for instance, an object representing the device is derived from the abstract system class and helps to seamlessly integrate it into IVORY.

5 THE INFORMATION VISUALIZATION MODELLING LANGUAGE (IVML)

5.1 Scope

As already explained in Section 2, the purpose of the script language developed for IVORY is to configure and parameterize individual visual metaphors implemented by the plug-ins. Since VRML 2.0 is not sufficient to specify the topology of large graphs, we had to devise a proprietary extension called IVML. As an interpreted high-level language it enables users on the *power* level to elegantly describe individual visualization problems and to build fast prototypes. The language is open, object-oriented and features scene graphs. In order to describe the geometry and visual appearance of layouts VRML 2.0 code can be embedded into IVML.

Similar to VRML 2.0 the basic building blocks describing the scene are objects defined by fields. The following example outlines a typical IVML object:

```
DEF myObj SampleData {
 label "Hello world!"
 num (23+7)/sin(0.334)
 visual IVVisual {
  color <0, 0, 1> * settings.intensity
 }
 appearance INLINE "file:/vrml/lora.wrl"
 datapath system.scriptbase + "/dbase"
}
```

The object type SampleData is implemented by the corresponding IVORY plug-in. Thus, we do not impose any restrictions to the number of object types in IVML. Other features comprise inline arithmetic expressions and references (settings.intensity), which replace the route mechanism along with JavaScript.

5.2 IVML specific Nodes

In essence, the following IVML-specific nodes, whose hierarchy is presented in Figure 8, make up the extensions to VRML 2.0:

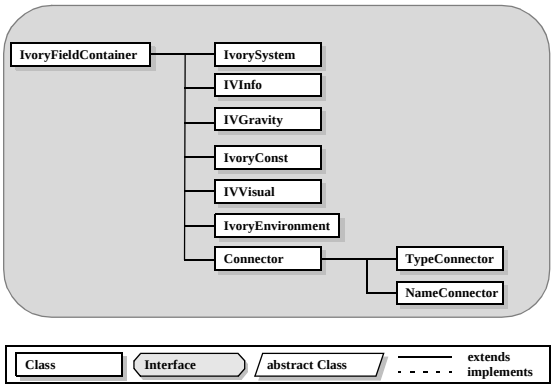


Figure 8: Hierarchy of IVML specific Classes

The header identifies the IVML script and version.

```
#IVML 1.0
```

Constants are represented as strings in `IvoryConst` objects.

```
DEF myconst IvoryConst {
 myTitle  "Hello World!"
 myURL    "http://hello.world.ch/basics.html"
}
```

System parameters are stored in an `IvorySystem` object, which is automatically initialized at parse time. It contains various fields, which are omitted here for brevity.

```
DEF system IvorySystem {
 cluster BaseCluster {
  visible TRUE
  transparency 0.8
 }
 layout BaseLayout {
  series IVTimeSeries {
   start 1.1.75
   stop 31.12.95
  }
 }
 info IVInfo {
  title "Economic indices"
  info  ["Example for Ivory V2.0"]
 }
}
```

Environment objects contain VRML inlays to define individual backgrounds, light sources, cursors and static scene parameters.

```
DEF env IvoryEnvironment {
 lights      INLINE "brightLight.wrl"
 cursor      INLINE "crossCursor.wrl"
 environment INLINE "financeEnv.wrl"
 billboards  INLINE "extendetBboard.wrl"
}
```

Connectors are objects which support the automatic description of the graph topology of the visualization problem. We distinguish between so-called `NameConnector` and `TypeConnector`. Both support wildcards to simplify the generation of object connections in very complex environments.

5.3 The Plug-in Interface

In order to reflect the plug-in mechanism, IVML features dynamic loading to invoke plug-in classes at runtime. It is similar to SGI's OpenInventor.

The following example illustrates the idea. Here, we define a simple, static graph consisting of 6 data objects and 12 interconnections. The associated connectivity matrix is given by:

Table 1: Connectivity matrix of the example shown below

	T1	T2	R1	R2	R3	R4
T1			1	1	1	1
T2			1	1	1	1
R1	1	1		1		1
R2	1	1	1		1	
R3	1	1		1		1
R4	1	1	1		1	

The IVML code fragment depicted below gives the full definition of the graph. We start with a prototype definition of a generic data object from which all subsequent instances are derived. The prototype contains a default initialization of information assigned to all objects, such as color or scale factors. Note that the geometry of the prototype object is hardwired in the plug-in `SimpleData`, which itself is derived from the BaseData class (see also Section 4). Next, two individual objects (ellipses in Figure 9) are instantiated overwriting some of the default parameters of the prototype. In particular, we overwrite the object appearance by an VRML 2.0 expression, which can be either inline or URL. Likewise, we generate the four spherical objects. The subsequent prototype definition of connection objects is used by the following expression, which describe the links between all objects. It can be seen, that the wildcard expressions in IVML, such as "R*" tremendously simplify the definition of the graph topology of our example.

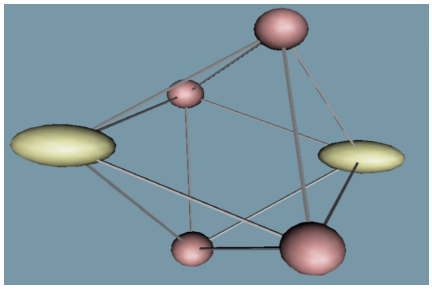


Figure 9: Layout computed from IVML script presented above

```
#IVML V1.0
# Example: Simple Objects

# First set some global parameters
IvorySystem {
  info IVInfo {
    title "Simple Objects"
    info  ["Ivory V2.0 Example"]
  }
}

# Define a prototype for our
# data objects
PROTO data {
  SimpleData {
    label "Simple Object "+me.name
    spaceFactor 5.0
    visual IVVisual {
      color <1,0.5,0.5>
    }
  }
}

# Define the 2 Top-Objects special
DEF T1 data {
  visual IVVisual {
    color <1,1,0.5>
  }
  appearance INLINE “ellipse.wrl”
}
DEF T2 data {
  visual IVVisual {
    color <1,1,0.5>
  }
  appearance INLINE “ellipse.wrl”
}

# Define the ring objects
DEF R1 data {}
DEF R2 data {}
DEF R3 data {}
DEF R4 data {}
```

```
# Define a prototype for our
# connection object
PROTO conn {
  SimpleConn {
    defaultLength 10.0
  }
}

# make the connections
# connect first top with all
# objects in ring
NameConnector{
  template conn {}
  names ["T1","R*"]
}

# connect second top with all
# objects in ring
NameConnector{
  template conn {}
  names ["T2","R*"]
}

# connect the ring objects
conn {
  label "R1_R2"
  leftObj R1
  rightObj R2
}
conn {
  label "R2_R3"
  leftObj R2
  rightObj R3
}
conn {
  label "R3_R4"
  leftObj R3
  rightObj R4
}
conn {
  label "R4_R1"
  leftObj R4
  rightObj R1
}
```

6 IMPLEMENTATION ISSUES

The IVORY implementation takes full advantage of all sophisticated features provided by Java 1.1. Each plug-in is mapped onto a Java class. Thus, the loading of individual plug-ins at runtime can be accomplished by the dynamic class loading mechanism of Java. The Java reflection model is used by the IVML interpreter to check, read and set the field values of plug-in objects.

We use the naming convention for setter and getter methods of Java Beans to access the underlying Java member variables. The Beans compliance enable to use all advanced Beans features, such as property editing. Although our current client-server implementation is based on a proprietary object serialization protocol, we are currently working on an RMI-based method invocation.

One of the critical implementation issues of IVORY was (and still is) the 3D API. The OpenGL bindings, available in beta-version from [22], are fast, however, are on a low abstraction level and do not support advanced object management by scene graphs. In addition, some rendering features, such as texture mapping are not implemented. In addition, VRML parsing has to be provided by the user.

Unlike OpenGL, Sun’s Java3D [23] provides a powerful API for 3D graphics including scene graph optimization and VRML extensions. The platforms comprise SUN and Windows 95/NT, however, the API has not been available during the implementation of IVORY.

Therefore, an early version of IVORY was based on Dimension X’s Liquid Reality class library [25], which was the only appropriate 3D Java API at that time. The robust beta version supported VRML and has been ported onto many platforms. Since the scene graph data structures are maintained in Java, the system performance is low.

Our current implementation uses SGI’s CosmoPlayer [24] for visualization. Unlike the libraries from above, the software is essentially a VRML 2.0 viewer plug-in for WWW browsers which has no immediate 3D API for Java. However, viewer control and callback-functions can be invoked through the LifeConnect mechanism of the WWW-browser and the External Author Interface (EAI) of CosmoPlayer. The scene rendering is based on OpenGL and thus supported by a wide range of hardware accelerators. At this time CosmoPlayer is available for SGI, Windows 95/NT and Apple Macintosh.

7 EXAMPLE

In physics-based visualization environments force-feedback provides a natural human-computer interface and mediates an additional sensoric cue to the user. Therefore, we invoked a Phantom® as 3D haptic interface. The Phantom enables to pick individual objects in 3D and to pull or push to “feel” the connection strength of objects for a given graph layout.

a)

b)

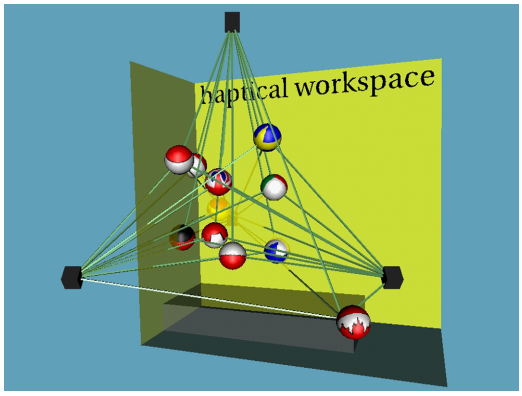


Figure 10: a) User sitting in front of the haptical device. b) View on the virtual representation of the haptical workspace. The layout shows the influence of four economic indicators onto the long term interest rates of different countries; Canada is currently being dragged by the user.

Further examples can be seen on our IVORY project homepage [26].

8 CONCLUSIONS AND FUTURE WORK

We presented a new approach towards a portable, object-oriented framework especially designed for physics-based information visualization. The system is open and expandable by adding new plug-ins. With the 4 layered abstraction model for users we provide adequate interfaces to configure our system at different abstraction levels. This covers a fast visualization prototyping using predefined plug-ins, but also a very flexible low-level system access. We also introduce a new script language named Information Visualization Modeling Language (IVML), which is distinctively proposed to describe information visualization problems. Future work has to encompass the development of hierachi-

cally organized layout subsystems, which make use of the object clustering analysis embedded into the system. We expect a tremendous speedup for the layout. Secondly, a widely strewn usability test including daily-business cases in the areas of our cooperation partner will validate IVORY's performance in practice.

ACKNOWLEDGMENT

This research has been made possible by the IT-Camp of the Swiss Bank Corporation (SBV), Basel, Switzerland. The authors thank T. Strasser and D. Bielser for implementing parts of the software.

REFERENCES

[1] Bruce H. McCormick, Thomas A. DeFanti, Maxine D. Brown, Ed.: *Visualization in Scientific Computing (ViSC)*. In Computer Graphics, ACM SIGGRAPH, Vol. 21, Number 6, Nov. 1987.

[2] *AVS Homepage*, Advanced Visual Systems Inc., http://www.avs.com.

[3] *IRIS Explorer Users Guide*, Silicon Graphics Inc., Mountain View, CA, 1991.

[4] *Data Explorer Reference Manual*, IBM Corporation, Armonk, NY., 1991.

[5] William J. Schroeder, Kenneth M. Martin, William E. Lorensen: The Design and Implementation *Of An Object-Oriented Toolkit For 3D Graphics And Visualization*, Proceedings IEEE Visualization '96, pp. 93-100, 1996.

[6] Robertson G. G., Mackinlay J. D., Card S. K.: *Cone Trees: Animated 3D Visualization of Hierarchical Information*, Proceedings Human Factors in Computing Systems CHI '91 Conference, New Orleans, LA, 1991, pp. 189-194.

[7] Munzner T., Hoffmann E., Claffy K., Fenner B: *Visualizing the Global Topology of the mbone*, Proceedings Symposium on Information Visualization 1996.

[8] I. Bruss, A. Frick: *Fast Interactive 3-D Graph Visualization*, Proceedings of Graph Drawing 95, Springer Verlag, LNCS 1027, pp. 99-110, 1996.

[9] M. H. Gross, T. C. Sprenger, J. Finger: *Visualizing Information on a Sphere*, Proceedings of the IEEE Information Visualization 97, pp. 11-16.

[10] R. Hendley, et al.: *Case Study - Narcissus: Visualizing Information*, Proceedings of the IEEE Information Visualization 95, pp. 90-96, 1995.

[11] S. Card, S. G. Eick, N. Gershon: *Information Visualization*, SIGGRAPH 96 Course Notes 8, 1996.

[12] Keim D. A.: *Visual Techniques for Exploring Databases*, Invited Tutorial, Int. Conference on Knowledge Discovery in Databases (KDD'97), Newport Beach, CA, 1997.

[13] *XGobi, a System For Multivariate Data Visualization*, AT&T Labs Research, http://www.research.att.com/~andreas/xgobi.

[14] Ahlberg C., Wistrand E.: *IVEE: An Information Visualization and Exploration Enviroment*, Proceedings International Symposium on Information Visualization, Atlanta, GA, 1995, pp. 66-73.

[15] MineSet: Silicon Graphics data mining and visualization product, Silicon Graphics Inc., Mountain View, CA, http://www.sgi.com/Products/software/MineSet.

[16] T. C. Sprenger, M. H. Gross, A. Eggenberger, M. Kaufmann: *A Framework for Physically-Based Information Visualization*. Eight EuroGraphics-Workshop on Visualization in Scientific Computing, France, pp. 77-86, 1997.

[17] *The source for Java Technology*, SUN Microsystems Inc., Palo Alto, CA, http://java.sun.com.

[18] *Home of the latest technologies from IBM*, IBM Corporation, http://www.alphaWorks.ibm.com.

[19] *TowerJ High Performance Native Java*, Tower Technologies, Austin, TX, http://www.towerj.com.

[20] *The VRML Consortium*, VRML Consortium Inc., http://www.vrml.org.

[21] A. M. Tannenbaum, Y. Langsam, M. J. Augenstein: *Data Structures Using C*, Prentice-Hall, 1990.

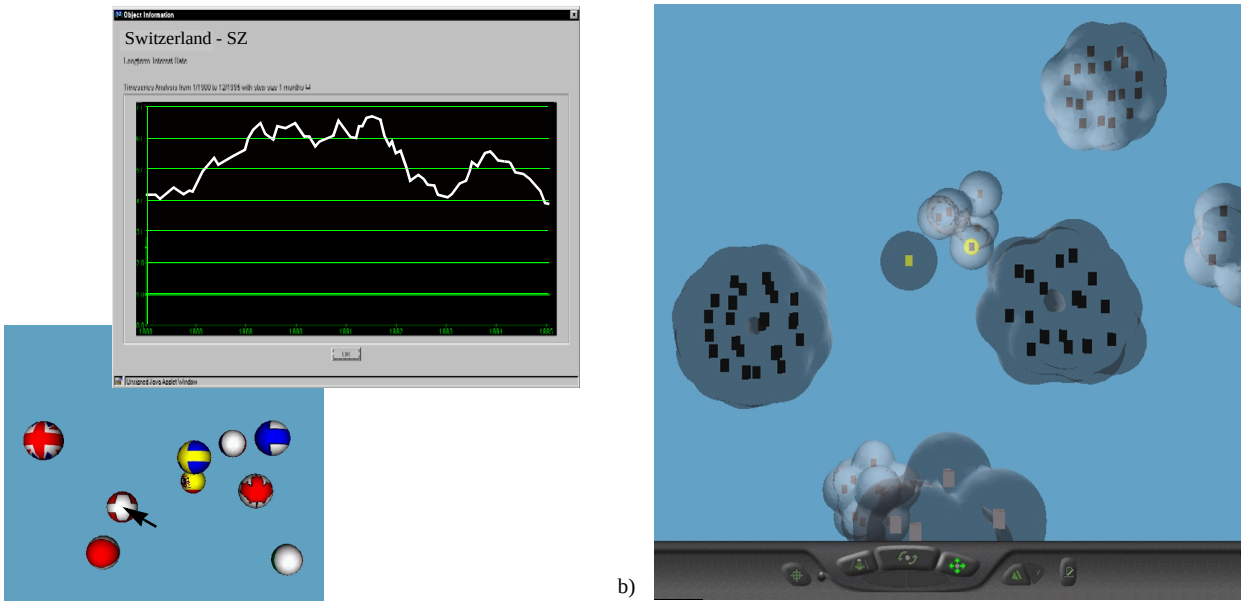
[22] *The Magician OpenGL Interface*, Arcane Technologies Ltd, http://www.hermetica.com/technologia/java/magician.

[23] *Java3D API Homepage*, SUN Microsystems Inc., Palo Alto, CA, http://java.sun.com/products/java-media/3D.

[24] *Cosmo Software Homepage*, Silicon Graphics Inc., Mountain View, CA, http://cosmosoftware.com/products/player.

[25] *Dimension Xs Liquid Reality*, Microsoft Corporation, Redmond, WA, http://www.microsoft.com/dimensionx/lr.

[26] *CGRG Homepage*, ETH Zürich, http://www.inf.ethz.ch/department/IS/cg/html/research/infovis.html.



a) b)

Figure CP-1: Results of possible user interactions. a) “Drill-through” the data provided by a currency plug-in. b) 3D Arrangement of a HTML-page domain (similarity criterion is the URL) after invoking the Blob-clustering algorithm from [9].

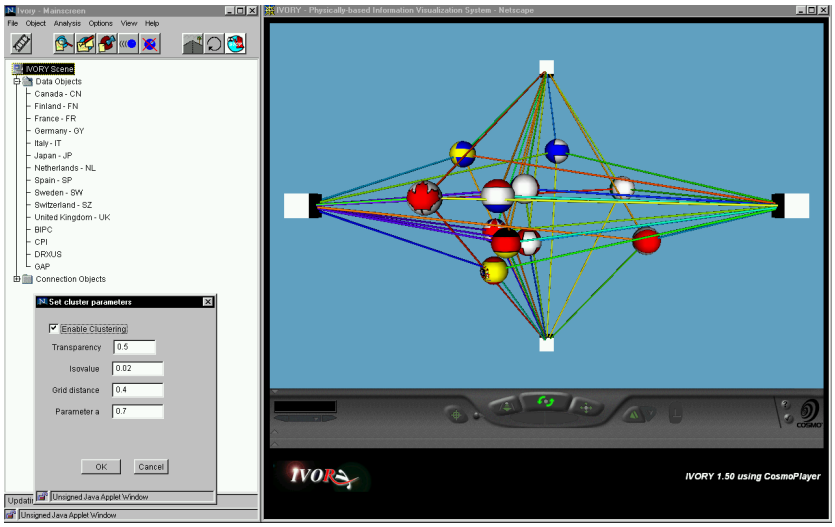
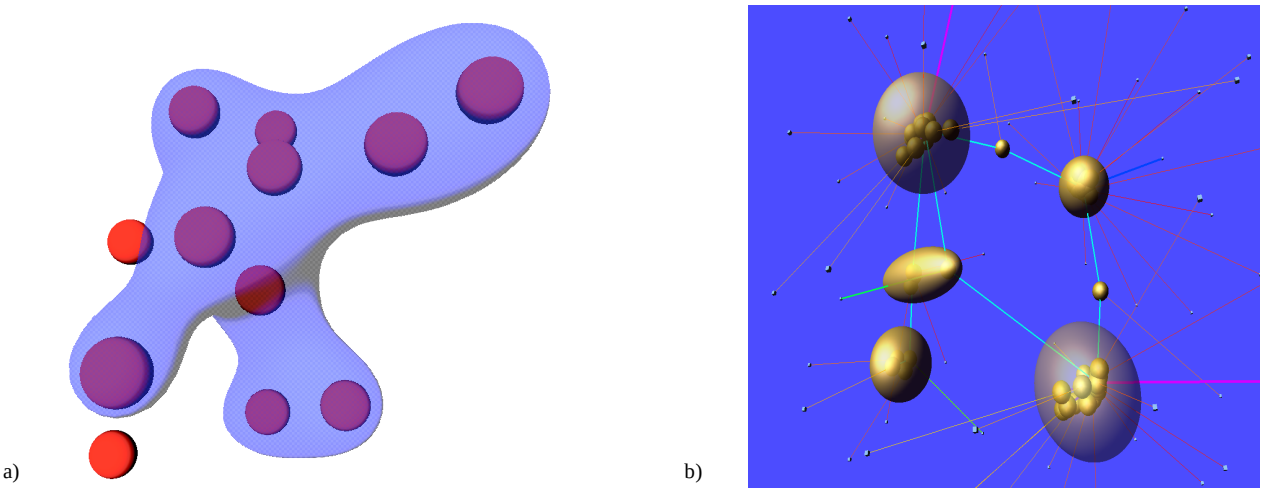


Figure CP-2: Screenshot of the IVORY frontend running on a Windows NT 4.0 machine under Netscape 4.04 with AWT 1.1 support.



a) b)

Figure CP-3: Two implemented types of clustering. a) Blob Cluster and b) Ellipsoid Cluster