MAMP: A Design Model for Object-Oriented Visualization Systems

Ricardo Orosco* & Marcelo Campo**

* Univ. Autonoma de Madrid – UAM – Dep. Ing. Informatica. Ciudad Univ. Cantoblanco, 28049, Madrid, Spain. Email: orosco@acm.org . Also UNCPBA – ISISTAN –Argentina

** Univ. Nac. Centro Bs. As. – UNCPBA - Fac. Cs. Exactas - ISISTAN. 7000, Tandil, Argentina. Email: mcampo@exa.unicen.edu.ar.

Abstract: The difficulty to build interactive visualizations that allow complex data explorations is a well-known fact. Some reasons for this difficulty are the dynamic nature of data exploration process and the diversity of goals and requirements in visualization applications, among others. The usage of design models for implementing information visualization systems (IVS) appears as a convenient approach to reduce this complexity. However, current HCI design models are insufficient from the point of view of the needs of IVSs. In a similar way, tools or environments specially developed to build IVSs have limitations for the implementation of highly dynamic exploration process. In this work, MAMP, a design model for the construction of object oriented visualization applications is presented. This model combines the advantages of current HCI design models (in particular, MVC) with the requirements of data visualization process. A software architecture and an object-oriented application framework, Telescope, based on MAMP are also described, along with several ways for implementing the characteristic features of any visualization, using these tools. Finally, two visualization applications developed according to the MAMP design model are briefly described: *CityVis*, a system for visualizing city data, and *WarVis*, a system for visualizing information about military conflicts.

Keywords User interface architectures, visualization, object-oriented frameworks

1. INTRODUCTION

This paper is devoted to the description of a design model for visualization systems construction, **MAMP**, and an associated software architecture that can be used

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35349-4_22](https://doi.org/10.1007/978-0-387-35349-4_22)

S. Chatty et al. (eds.), *Engineering for Human-Computer Interaction*

© IFIP International Federation for Information Processing 1999

to simplify the development of Information Visualization Systems, IVS. A corresponding Object-Oriented framework, *Telescope*, has been built and tested through the development of visualization systems with a sophisticated functionality in different domains.

The difficulty to build interactive visualizations that allow complex data explorations is a well-known fact (Chuah 96)(Kazman 96)(Roth 94). There are some reasons for this intrinsic difficulty: on one hand, advanced visualization systems require highly dynamic data exploration processes, in which data are subject to successive transformations using composition, decomposition and filtering mechanisms, as well as different degrees of abstraction and detail are needed. The decision about the use of one of these operations is taken by the user on the basis of the result of the exploration process itself, in an unpredictable way. The exploration of the same data with different goals, at different moments or by different users, can rise the need for different exploration techniques, so a framework for the development of IVSs must simplify the adaptation of the system to the users needs. On the other hand, data also shows a dynamic behavior, and very often the information conveyed by it can not be predicted when the IVS is developed. Finally, IVSs are usually conceived to be used in a number of applications related to a specific field, but there are always differences between the requirements of different applications, so the exploration techniques being used have to be adapted frequently. The development of an IVS is generally a process based on successive approximations, where the reuse of techniques from previous systems is essential, and the availability of a clear model can make a big difference.

One of the consequences of the above facts is that the development of IVSs is in general very different from the development of generic interactive applications for which significant portions of their interactive functionality can be built using modern interactive graphical tools like interface builders. In fact, the development of IVSs requires in general a big amount of programming, and a lot of redesign and adaptation of critical parts. In this context, the most powerful support that can be given to the design and development of IVSs is through a simple conceptual design model that simplifies the designer the decomposition of the system into different components, and their integration, reuse, and adaptation.

Finally, one of the main facts that differentiate IVSs is what we can refer to as their *visualization functionality*, that is, the different ways they allow the user to manipulate data visualization. As a matter of fact, the appearance of data flow based visualization systems, like AVS (Upson 89), can be considered as a major step in the simplification of the design and use of IVSs. The work presented here puts together ideas coming from this recent trend with others from HCI modeling, allowing the use of techniques that can already be seen as classical within the HCI context, but that are clearly insufficient from the point of view of the needs of IVSs, and at the same time giving more flexibility to the techniques of data flow based visualization.

The model proposed in this paper consists of an extension to the *Model View Controller* software architecture (MVC) (Krasner 88), adding an Abstraction component and a Mapping component. These new components receive requests from visualization component, and return elaborate data representations after consulting the real data component. On the other hand, abstraction components can be organized in

several ways, allowing the implementation of highly dynamic visualization systems, with a visualization functionality that depends very heavily on the application state.

The rest of this paper is organized as follows: in §2 current paradigms for HCI modeling and their limitations for implementing IVS are described. Section 3 describes MAMP design model, its derivation from MVC and the advantages of its usage. In §4 and §5 a possible implementation for this model is described, through the Telescope framework. Section 6 describes the construction of some of main visualization features, using Telescope. Section 7 describes (briefly) two visualization applications developed with this model. Finally, §8 cites some conclusions of this work.

2. MODELS, ARCHITECTURES AND TOOLS FOR VISUALIZATION DEVELOPMENT

Current HCI design models provide essentially the same functional capabilities: presentation, application and dialogue control. These models addressed the same quality goals, but differences in how these goals were applied led to two main architectural styles: the object-oriented style, as exemplified by MVC and PAC (Coutaz 87), and the layered style, first exemplified by the Seeheim model (Green 85) and then followed by the Arch/Slinky meta-model (UIMS 91).

Within object-oriented style, MVC is the most common and used model. In particular, in object-oriented visualization systems with direct-manipulation capabilities, MVC is the predominant approach. This model promotes the independence between the data representation (Model) and its visual presentation (View), assigning to the View component the responsibility of producing the visual presentation of data contained in its Model.

From the point of view of a reusable architecture for IVS, MVC fails in providing an adequate architectural guide that allows the independent incorporation of additional functionality. Indeed, to provide an adequate support for tasks such as exploration process control and semantic processing of data, the identification of additional components should be considered. Particularly, some of the major limitations that can be identified are:

- *Semantic Analysis.* From a design perspective, MVC does not promote the implementation of mechanisms of semantic analysis of the information independently of classes representing information or views. That is, data analysis algorithms must be located in classes implementing either the information representation or visualizations. When new algorithms need to be added, this solution forces the modification of existent classes with behavior related to specific applications. In this way, MVC can induce designs hard to extend and adapt.
- *Abstraction levels filtering.* In general, complex information defines several levels of abstraction that can be visualized or hidden according to the detail level desired by the user. Using a pure MVC architecture, the filtering of different detail levels of the data to be visualized has to be implemented for any particular visualization.

This complicates the programming of alternative visualizations, reducing also its reusability.

- *Data enhancement / enrichment processes.* The usage of some visualization techniques can require a process of data enrichment or enhancement over the data to be visualized, such as an interpolation process. However, the process to be applied depends both on the selected visualization technique and the visualized domain, in order to avoid undesirable phenomena such as misrepresentation (Duce 93). The implementation of these processes by means of the distribution of their functionality among the views and model classes, reduces the possibilities of reuse of these components, and limits the usage of different processes.

The layered style has similar goals, although its emphasis is in the portability of application components across different domains and visualization toolkits. It presents similar drawbacks to the ones cited for the MVC approach. For example, in Arch/Slinky, the features previously described are generally distributed among the dialogue component, the functional core adapter and the presentation component.

The usage of specific tools or environments for visualization construction also presents some limitations for implementing visualizations with a high functionality. Some of these tools attempt to alleviate the construction task by means of techniques and methods for automatic generation of visualizations (BOZ (Casner 91), APT (Mackinlay 86)). This approach facilitates the construction of presentations by non-programmers. These tools, however, are focused on the determination and construction of adequate presentations for particular data types and tasks, without considering aspects such as the adaptation of visualization techniques to specific application needs, or the inclusion of visualization functionality. Another approach is to use special-purpose toolkits and libraries, specially designed for visualization construction (OpenGL (Neider 93), Visualization Toolkit (Schroeder 96)). These tools attempt to provide a high performance, but the degree of abstraction provided by them is far away from the needs of designers of complex visualization.

One successful approach, in particular in commercial visualization communities, is to use dataflow-based systems or environments (such as AVS). These environments have a modular approach for visualization design. The designer must compose a particular visualization through the construction of a network consisting of predefined modules (which include different visualization techniques, filters and operators), generally contained in libraries. This approach has a wide acceptance, although its applicability is mainly oriented to scientific visualization applications. However, it also presents some limitations for implementing visualizations with complex functionality. In most cases, designed networks have a static nature that makes the implementation of highly dynamic visualizations difficult. Additionally, providing an adequate support for the implementation of dynamic data exploration processes, such as the incorporation of domain-specific tasks or user-assistance techniques, is a complex task.

The usage of a software architecture for visualization construction is an approach that has not been sufficiently explored. VANISH (Kazman 96) is one of the few examples in this area. This architecture adapts the Arch/Slinky metamodel for building visualization applications of hierarchically structured information. In this work, a similar approach is pursued, using an object-oriented approach instead of a pure layered approach, incorporating the specific features of visualization processes.

3. PROPOSED MODEL

In order to provide a design model for the structuring of object-oriented visualization applications, MVC was selected as the base architecture model, mainly because its wide acceptance in developing interactive object oriented applications. To solve the previously described limitations of this model, the specific features of visualization processes were considered from the perspective of a conceptual model of visualization process, *Visualization Idioms* (Habber 90). This model focuses on the description of a data visualization process, but it does not consider very important aspects such as user interaction. Its integration with MVC allows to obtain a domain specific software architecture for implementing object oriented visualization applications with highly dynamic capabilities, as it is required to perform successful data exploration processes.

Figure 1. Visualization Idioms model

The *Visualization Idioms* model identifies the three main transformations occurring in most visualization processes (fig. 1), that convert raw data (source data to be visualized) into a displayable image:

- *Data enhancement or enrichment*: it operates on raw data, which is modified into derived data for subsequent visualization operations.
- *Visualization Mapping*: it constructs an imaginary object (abstract visualization object – AVO) from the derived data produced by data enhancement transformations. This imaginary object has extensions in space and time, and contains attributes such as geometry, color, transparency, luminosity, etc.
- *Rendering*: it operates on the AVO to produce at least a displayable image, generally using familiar operations from computer graphics and image processing.

All of these transformations can be implemented using standard MVC. For example, in most object-oriented applications based on MVC, the rendering transformation is included in the views, while other transformations and raw data administration are included within the model. Taking into account the limitations described in §2, this fact produces designs hard to understand and with little reusability.

To obtain a better design, the MVC's model component was divided into three new components, considering those transformations described by *Visualization Idioms*:

- *Data Representation or Application Model*: it manages the raw data to be visualized, that is, the analyzed information. Generally, these data can be available in several forms: data contained in a database or gathered from another application.
- *Visualized Data Determination or Data Abstraction*: it determines which data will be effectively visualized at run-time. To do this, it uses the data provided by the application model, and determines the information that will be presented to the user taking into account aspects such as: visualization state (selected items, current zoom level, user actions, etc.), user preferences, user directives, detail levels, etc. This component implements the specific functionality of particular visualization applications; and it is described with more detail in §4 and §5.
- *Data Objects – Graphical Objects Association or Mapping*: it performs the association between visualized data objects and their correspondent graphic representation. This association consists of a tuple detailing the type of graphical object used to represent each item. It also contains associations with the form <visual channel, data attribute> indicating the visual channels used to show some properties of data provided by abstraction level.

According to this subdivision of MVC's model component in three new components, a new design model, specially oriented to the construction of object-oriented visualization applications can be derived: **MAMP (Model-Abstraction-Mapping-Presentation)** (fig.2). The first three components correspond to the ones above described, while the Presentation component includes the functions of MVC views and controllers. However, this last component can be further subdivided into two components: representation and interaction, according to the application needs.

The addition of these new components facilitates the construction of visualization in several ways. First, it is possible to build different visualization applications of the same dataset, using the same visualization techniques, but with different functionality. With this design, the view and application components are fully reused; the designer only needs to specify the tasks to be performed, the abstraction levels to be used for visualizing the data, the selection criteria, the analytical algorithms to be applied over the data, and so on. In this way, the designer may build systems that perform different exploration processes of the same data.

Figure 2. MAMP model: components and relationships

Also, it is possible to reuse different data enhancement or analysis algorithms in visualizations of distinct domains. The visualizer may explore the data through different analysis, using algorithms with different characteristics. These algorithms can be stored in a library, from which the designer can select them for other visualizations. Obviously, it is also possible to build visualizations of the same domain with different visualization techniques, or to use the same visualization techniques in different domains.

The separation of the mapping component from the application model allows encapsulating the knowledge about the presentations used for each data item. Thus, the representation of each data item is determined according to the mapping specification, independently from the model, the visualization functionality and the visualization techniques. This means that the same mapping specification could be used with different visualizations.

Application-specific tasks and/or user-assistance techniques can be easily included in the design of the application. This feature allows the adaptability of the application to the domain or applications needs, and to the user preferences or expertise degree. Abstraction and Mapping components can include user or interaction models that tailor information presented to the user.

An additional feature of this approach is the possibility to dynamically change the application functionality. The algorithms and techniques can be replaced by similar ones at run-time, according to the user directives. The possibility to dynamically change the visualization functionality is a powerful feature for IVS. In this way, it is possible to build a visualization application that can be dynamically adapted at run-time to the visualization process performed by the user. This means that the behavior of the application can be modified, according to the user or application requirements. Also, it allows to the user perform fully dynamic exploration processes, doing different analysis of the data during his interaction, without rebuild the application.

It can be argued that some of these features can also be implemented using standard MVC, without explicitly including the abstraction component. But, the reusability of such designs would be very poor, because of the inclusion of application-specific knowledge in only two components. The inclusion of this knowledge as separated components facilitates its reusability, as well as enables designs with a higher flexibility and generality.

4. ABSTRACTORS

The described model admits different alternative implementations. In this work, the concept of *abstractor objects* to implement the abstraction component is introduced, emphasizing dynamic object composition to increase the reusability of the solutions. These objects are entities that can access the data representation (contained within the application model), and determine which objects (and properties) will be effectively visualized. To perform this determination, abstractors encapsulate the specific functionality of visualization applications, such as analysis algorithms, selection criteria, data enhancement/enrichment techniques, user assistance techniques, user or interaction models, etc.

Essentially, abstractors behave as *proxies* of the data models, controlling the access to these objects from visualization classes. In this way, abstractors can be composed to provide independent control over each object contained in the model. Also, they can be dynamically composed to combine different functionality, as for example, filtering on a specific selection of the visualized information.

Following the conventional communication mechanism among views and models in MVC, MAMP views request their models the information to be visualized through direct messages. In MAMP, an abstractor plays the role of a MVC model and decides whether to ask the original MAMP model for the requested information. The key idea behind the design of the abstractor support is the definition of a standard protocol, through which visualizations can ask their models for data to be visualized. Each view must ask this information through standard messages, such as *getItems*, *getNodes*, *getLinks* and so on.

Also, making the existence of abstractions explicit, abstractor allows to implement a mechanism to control the abstraction or detail level of the visualization, through the concept of *abstraction scales*, as explained in section 6.

The implementation and usage of abstractors using an object-oriented application framework is described with more detail in next sections.

5. TELESCOPE FRAMEWORK

MAMP is currently implemented by an object oriented framework, named *Telescope* (Orosco 97a), that materializes a software architecture for *MAMP*. *Telescope* was implemented in *Visual Works – Smalltalk*, using OpenGL as 3D graphics platform. Its main components are:

- **Data Representation:** storage and access to the data items to be visualized.
- **Data Abstraction:** this level determines which elements will be visualized, and classifies them into sets of related items.
- **Data Objects - Graphical Objects Mapping:** this component performs the association between the data objects (and their properties) and the correspondent graphical objects.
- **Visualization:** it comprises graphical data presentation and user interaction. It is sub-divided into two components:
 - **Presentation:** it consists of presentation building, according to the graphical objects specified in the mapping.
 - **Interaction:** it consists of all aspects related to the user interaction: item selections and manipulations, user navigation, etc.
- **Visualization State:** it contains the current visualization state, which can include the current selected items, current visualization focus, previous user actions, etc.

Each *Telescope*'s component is implemented by means of classes (generally abstract classes) providing its generic behavior. Each visualization application developed using *Telescope* must implement the particular behavior for each component. Fig. 3 shows relationships among the main classes of each component.

The visualization component consists of a set of *Views*, hierarchically organized. Each view contains a graphical presentation (an instance of a *GraphicalObject*

Figure. 3. Telescope Object Model

subclass), and an interactor for each allowed operation on it (for example, a *SelectInteractor* is used to perform item selections).

Views have an associated *Mapping* that performs the conversion between data objects and graphical objects. These mappings can be customized by the user at run-time or modified by the system according to changes in the visualization state. The *Mapping* class can be specialized through subclasses, for example to implement distortion-based visualization techniques.

Additionally, each view gets the information to be displayed through an associated model. This model consists of an abstractor (instances of an *Abstractor* subclass), which determines objects and properties to be effectively visualized. This abstractor has a reference to the data object, which contains its source data.

The visualization state component is mainly implemented by means of a *VizState* class, which manages all the information related to the current state of the visualization process.

6. ABSTRACTORS USAGE

In this section, some of the main visualization features that can be implemented by designers are shown, by defining different kinds of abstractors that can be used in *Telescope*, as well as a hierarchy of abstractors that defines a complex data abstraction policy. The *Telescope* framework includes a predefined set of general purpose abstractors that can be used or adapted for the application designer in order to implement his visualization applications.

In the general way, views requests to the abstractor the items to be visualized, through interface methods provided by the abstractor. For example, in most visualization techniques implemented in *CityVis* and *WarVis*, the communication between views and abstractors consists on a *getItems* method, that provides to the view the data items to be visualized. Other visualization techniques can require a different interface; for example, graphs ask for their nodes and links through *getNodes*

and *getLinks* methods. Thus, views do not need to worry about how to determine which data will be presented to the user; they only need to manage the presentation aspects, such as the construction and rendering of presentations.

6.1 Filters

The simplest and most frequently used kind of abstractor is a filter. A filter simply selects a set of items from the application data model according to a given condition. The filtering condition can be fixed or it can be determined dynamically, for example depending on the users' requirements during the visualization process. The general form of a filter is

```
getItems
  items := subject selectItems: #objectType
  filtered_items:=items select: [:it | it satisfies: condition ].
  ^filtered_items
```

This code shows the typical form (in a very simplified way) of an abstractor interface, which generally consists on a method returning the items to be visualized. Here, the method *selectItems* retrieves the data from the storage, and then selects those items that satisfy a given condition (through the method *select*); finally, selected items are returned (to its associated views).

A particular kind of dynamic filters that is worth remarking is characterized by the fact that the filtering condition is parameterized and the values of the corresponding parameters are determined by actions of the user, like selecting items belonging to a particular topic (see *CityVis* below, section .7.1)

6.2 Derivation of new information

A second type of abstractors corresponds to the addition of new information to the visualization data. This information can be of several kinds, among which some examples are:

- *Derived information*: this kind of information is usually the product of an analysis algorithm or some other similar mechanism, as the use of statistics about the available data,

```
getItems
  items := subject selectItems: #conflicts.
  stats := StatisticsInfo computeDistributionOf: #fatalityLevel in: items.
  ^stats.
```

Here, *getItems* computes the distribution of the variable *#fatalityLevel* in a set of objects representing military conflicts. This computation is performed by the *StatisticsInfo* class, through the method *computeDistributionOf:in:*. *Telescope* provides a wide set of classes and methods for the calculation of statistics, metrics or similar measures, from which *StatisticsInfo* is only an example. In a similar way, the user may implement specific classes and methods for deriving new data, according to application needs.

- *Data enrichment*: Algorithms for data enrichment or enhancement, like data interpolation, can also be included in abstractors. They can include domain

information to avoid undesired facts like misrepresentation, although this can reduce their reusability.

```

getItems
  discrete_data := subject selectItems: #discreteTemperatureMeasures.
  continuum_data := self interpolateData: items withConstraints: (self
    domainConstraints).
    ^ continuum_data.

```

Here, the abstractor retrieves a set of values for temperature measures, for certain discrete locations. These data are then interpolated to obtain a set with continuous values, which are adequate for some visualization techniques, such as plots. The interpolation process takes into account some constraints imposed by the domain (e.g. temperature values cannot be higher than 100°C), to avoid incorrect measures.

- *Data aggregation (composition/decomposition)*: very often the desired visualization is based on groups of data more than on individual pieces of data. For example, we might want to be able to show a world map, containing only information about continents and not about countries. If the data available is only about countries, it is necessary to aggregate the information of countries belonging to the same continent in one object.

```

getItems
  countries := subject selectItems: #countries.
  continents := self deriveContinentInformationWith: countries.
    ^ continents.

```

This aggregation process can also be controlled by the user during his interaction, allowing different ways to build groups of data, different kinds of objects to which the accumulation process is applied, etc.

6.3 Abstraction Level Management

Abstractors allows the automatic management of abstraction levels through the usage of abstraction scales. An abstraction scale is an ordered tuple naming the order in which information, like a city map for example, should be visualized. A scale can have its own user-interface control (e.g. a slider) through which the user can interactively vary the level of abstraction of the visualization (i.e. showing or hiding dynamically details). Visualizations, in turn, only have to worry about what must be shown according to the data that abstractors pass to them, in the current abstraction level. For example, the scale below can be used to define the different detail levels in which a city map can be shown.:

```
(regionalView, globalView, detailedView, streetView)
```

A selection of a level in this scale will define which information the visualization will receive to be graphically presented. That is, if the selected abstraction level is *globalView* the visualization will only receive the information about roads, main streets and main interesting points in the city. After that if the user selects *detailedView*, the same visualization will receive the same information plus the information about other minor city streets.

To implement abstraction scales, each abstractor can have associated one instance of *AbstractionLevel*, which defines its current abstraction level and implements the abstraction scale mechanism. Subclasses of this class provide the support for

comparing symbolic levels of abstraction according to the type of scale (i.e. discrete scale, inclusive scale, continuous scale, etc.). This scale can be interactively manipulated by the user to dynamically increase or decrease the detail level of the visualization, through the use of controls such as sliders or radio buttons.

6.4 Hierarchies of Abstractors

Atomic abstractors like the ones described up to here do not suffice for usual applications. The usual situation is that several simpler abstractors have to be composed to allow the accumulation of their corresponding data or the composition of their functionality. A mechanism for the hierarchical composition of abstractors can be used for this, and also to achieve other complex effects. For example, an abstractor that is responsible to give the information needed for the visualization of a graph can delegate the computation of this information into several component subabstractors. A *GraphAbstractor* exemplifies a simplified version of this situation:

```
getNodeNodes
  "first subabstractor computes nodes of the graph"
  ^ (components at: 1) getItems
getNodeLinks
  "second subabstractor computes links of the graph"
  ^ (components at: 2) getItems
```

This allows to change dynamically the configuration of the *GraphAbstractor* at execution time, by just changing its component subabstractors. A common use of this feature is to change one subabstractor when the user produces a change in the current abstraction level. This allows the visualization of a graph with different abstraction levels, where, in each level, the nodes and links are computed in a different way. Thus, the designer can implement particular abstractors for computing nodes or links in each abstraction level. Upon each level change, the root abstractor can change some of its subabstractors, replacing the old subabstractors by the abstractors that compute the items in the new abstraction level. An additional advantage of implementing specific abstractors for each level, is that the abstractors can be independent of the abstraction scale used; that is, the same abstractor, representing a particular level, can be used in different scales containing this level.

6.5 Assistance Techniques

In the last years, several visualization techniques providing some degree of user assistance in the visualization process have been developed (e.g. *Galaxy of News* (Rennison 94)). *AutoFocus* (Orosco 97b) is an example of assistance technique, whose goal is to assist the user in the exploration process, providing help in the focus specification and using 3D presentations conveying more information. The assistance in the focus specification is implemented by a statistical inference mechanism, that attempts to detect the criteria used by the user in item selections. Once these criteria have been estimated by the system, item presentations are updated accordingly to them, reflecting the relevance of each item with respect to the currently defined focus (using an item opacity proportional to the item importance). In this way, a quick approximation to the desired final focus is obtained, facilitating the task and reducing

the user workload. Particularly, *CityVis* uses *AutoFocus* to assist the user in the searching of apartments on sale. This kind of techniques can be implemented through the incorporation of specialized abstractors, that perform the inference process and calculate each item relevance.

7. APPLICATIONS

In this section, two applications developed with *Telescope* are briefly described: *CityVis* (Orosco 97a), a system for the visualization of city data, and *WarVis* (Orosco 98), a system for the visualization of information about military conflicts. *Telescope* was also used in the development of other applications, such as *Luthier* (Campo 97a), an object-oriented software visualization system, and *MetaExplorer* (Campo 97b), a system for three-dimensional visualization of design patterns. All of these systems were implemented in *Visual Works - Smalltalk*, using *OpenGL* as 3D graphics platform.

7.1 CityVis

CityVis is an information visualization system for city data, with the goal to support the exploration of city information, providing techniques to assist the user in this process. The tool provides 3D visualizations of the city information. These 3D views can be converted into 2D views, by constraining the user navigation capabilities.

CityVis makes an extensive use of abstractors to provide powerful filtering mechanisms for semantic zoom based on inclusive abstraction scales. These mechanisms allows the user for the continuous zooming of city maps, showing or hiding pertinent information at each level of abstraction.

Interactive filtering of information through query capabilities is provided as an additional mechanism to reduce the complexity of visualizations. This capability is implemented by an abstractor that provides the views with those data resulting of the query evaluation. In this way, any visualization can reuse the generic query management mechanisms.

The information is organized in layers, allowing the differentiation of several topics in the visualization. The system visualizes different kinds of information associated to the city (hotels, apartments, restaurants, transportation, statistics, etc.), that may be combined by the user to build the desired visualization. These topics are managed by a hierarchy of abstractors, which is dynamically reorganized during user exploration.

The system provides several functions for the administration and management of information revealed during the exploration process, such as the definition of new topics or the computation of statistics about the visualized data. In both cases, specialized abstractors compute this new information, making it available for further explorations of the same city data.

Figure. 4- Different visualizations produced by *CityVis*

Finally, *CityVis* implements assistance techniques, attempting to alleviate the user's work in the exploration. *AutoFocus* is one of these assistance techniques, which is implemented as described in previous section.

Fig. 4 shows an snapshot of *CityVis*. This picture shows several visualization techniques available in the system: layered organization of presentations, management of abstraction scale (semantic zoom), different abstraction levels and statistics about the concentration of data items in the map.

7.2 WarVis

WarVis is another visualization system developed with *Telescope*. This system provides visualizations of data about military conflicts among countries, from year 1800 to the present. It contains similar features to *CityVis*, such as the provision of 2D and 3D presentations, and the immersive navigation capabilities. The system consists on a set of different visualization techniques (such as plots or graphs), which the analyzer may use to visualize the data. He can dynamically create new views of the data, through the selection of an appropriate visualization technique. These presentations can share some abstractors (or subabstractors), because *WarVis* allows to provide several visualizations of the same data (provided by the same abstractor).

WarVis also has several abstraction scales, which can be interactively managed by the user. These scales control the information presented to the user and the way this information is calculated. In this case, a dynamic hierarchy of abstractors is used;

Figure. 5. Visualization techniques in WarVis

upon each abstraction level change, the hierarchy is modified to reflect the new abstraction level.

This system also provides non-conventional visualization techniques, such as *Abstract 3D Information Spaces*, which can be tailored by the system designer to the application needs. In this case, a particular version of this paradigm, Temporal 3D Lines was implemented, to visualize the location of each military conflict in the time.

Fig. 5 shows several visualization techniques used in WarVis: 3D plots, 3D graphs, and temporal 3D lines. The information presented in each case is determined by the user, through the use of device controls, as it is shown in the same picture.

8. CONCLUSIONS

MAMP, a design model for object-oriented visualization systems was presented, along with a *Telescope*, an application framework based on it. Their suitability to the construction of visualization systems was demonstrated through the development of several applications for different domains and purposes: *CityVis*, for the visualization of geographical data (physical data with a given physical form); and *WarVis* for the visualization of military conflicts (combining data of abstract and physical nature); and *MetaExplorer* for the visualization of object oriented software (data without a physical form, with an abstract nature)

The inclusion of an explicit abstraction component, implemented by means of abstractor objects, enable the construction by composition of complex filtering mechanisms that greatly simplify the implementation of visualizations, which is often

the more time-consuming task in the development of visualization systems. Using *Telescope*, the development of new visualizations is simplified, due to the high reusability degree of the main components of the system. In our case, the development of *WarVis* was relatively fast, due to the previous components developed in *CityVis*. Additionally, the extensibility of these systems is high, due to the clear localization of possible changes and extensions. As another feature, *Telescope* allows the quick prototyping of new visualizations or techniques, due to its high flexibility.

The architecture preserves the modularity of dataflow systems, allowing the user to compose modules (abstractors) and views (visualization techniques), conforming a network. The produced network has a dynamic nature, allowing the addition and removal of new modules and/or visualization techniques at runtime.

Future work with *Telescope* includes development of an environment for visualization applications construction. This environment should automate part of the construction task, and guide the system-designer in the development of new applications or components. Also, the development of new visualization applications, particularly for WWW-based visualization and for numerical data domains are programmed.

9. REFERENCES

- Campo, M. Orosco, R. Teyseyre, A. (1997) Automatic Abstraction Management in Information Visualization Systems. Proc. Intl. Conf. IV '97, London, 08/97.
- Campo, M. (1997a) Compreensao Visual de Frameworks atraves da Introspecao de Exemplos. Phd. Thesis, UFRGS, Brasil (in Portuguese).
- Campo, M. Orosco, R. Teyseyre, A. (1997b) Interactive Abstraction Control in Visualization Applications. Proc. SCCC '97, Valparaíso, Chile, 10/97.
- Casner, S.M. (1991) A Task-analytic approach to the automated design of graphic presentations. ACM Transactions on Graphics. 10(3), 111-151.
- Chuah, M. Roth, S. (1991) On the Semantics of Interactive Visualizations. Proc. InfoViz '96, IEEE CS Press.
- Coutaz, J. (1987) PAC, An Implementation Model for Dialog Design. Proc. Of Interact '87, Stuttgart, September 1987, 431-436.
- Duce, D. A. (1991) Visualization. Proc. Visualization '93 Conf.
- Green, Mark (1985) Report on Dialogue Specification Tools. in G. Pfaff (ed) User Interface Management Systems, New York, Springer Verlag, 9-20.
- Haber, R.B., McNabb, R. (1990) Visualization Idioms: A Conceptual Model for Scientific Visualization Systems. Visualization in Scientific Comp., 74-93.
- Kazman, Rick. Carriere, Jeromy. (1996) Rapid Prototyping of Information Visualizations Using VANISH. Proc. IEEE InfoViz'96, San Francisco, CA.
- Krasner, G. Pope, S. (1988) A cookbook for Using Model-View-Controller interface paradigm in Smalltalk 80. Journal of O. O. Programming, 08-09/1988, 26-49
- Mackinlay, J. (1986) Automating the design of Graphical Presentations of Relational Information. ACM Trans. on Graphics, 5(2), ACM Press, 110-141.
- Neider, J. Davis. T. Woo, M. (1993) OpenGL Programming Guide. Addison-Wesley.
- Orosco, R. (1997) User Assistance in Information Visualization. Hypertext, Information Retrieval and Multimedia '97 Conf., Dortmund, Germany.

- Orosco, R. (1997a) Telescope: A Software Architecture for Visualization Systems Construction. Proc. ASOO '97, Buenos Aires, Argentina, 08/97.
- Orosco, R. Moriyon, R. (1998) Reducing the Effort of Building Object-Oriented Visualizations, 22th. Intl. COMPSAC'98 Conference, Viena, Austria, 08/98.
- Rennison, Earl. (1994) Galaxy of News: An Approach to Visualizing and Understanding Expansive News Landscapes Proc. UIST '94, 11/94, 3-12.
- Roth, S. Kolojejchick, J. Mattis, J. Goldstein, J. (1994) Interactive Graphic Design using Automatic Presentation Knowledge. Proc. CHI '94, 04/94, 112-117.
- Schroeder, W. Martin, K. Lorensen, B. (1993) The Visualization Toolkit, An Object Oriented Approach to 3D Graphics. Prentice hall, 1996.
- UIMS Tool Developers Workshop. (1991) A Metamodel for the Runtime Architecture of an Interactive System. SIGCHI Bulletin, 24(1), 1991, 32-37.
- Upson, C. (1989) The Application Visualization System: a computational environment for scientific visualization. IEEE CG&A, 07/1989

BIOGRAPHIES

Ricardo Orosco received a PhD degree in Computer Science by the Universidad Autónoma de Madrid (Spain). He is a research staff member at the Systems Research Institute, Tandil, Argentina, in the Objects and Visualization Group. His research interests include user interfaces, information visualization, and three-dimensional interactions.

Marcelo Campo received a PhD degree in Computer Science by the Universidade Federal do Rio Grande do Sul (UFRGS, Porto Alegre, Brazil). Currently is the leader of the Objects and Visualization Group, at the Systems Research Institute, Tandil, Argentina. His research interests include object oriented application frameworks, software architectures, design patterns and information visualization.

Discussion

Henrik Christensen: Implementation arrows point from GUI to Model. Does this imply that visualization is static with respect to the 'on the fly' changes in underlying data?

Ricardo Orosco: We currently assume that data are static. In the future we must consider this possibility. Future work must be done to cope with dynamic data.

Prasun Dewan: How is your work different from the Mastermind project of Pedro Szekely?

Ricardo Orosco: Several ideas from Mastermind were used in this project. Some of our colleagues were part of the Mastermind team. We got the idea of a mapping from his team.

John Grundy: Lots of 'difficult' problems to 'automate': interface to raw data/tools, abstraction of raw data, mappings between abstractions, specifying graphical form, translating interaction to a visual form. Which aspects are you trying to automate?

Ricardo Orosco: The goal of our work is to automate parts of the process. It is not possible to automate all. We try to provide guidelines for each reusable part.