Software Architectures for Designing Virtual Reality Applications

Conference Paper · May 2004

DOI: 10.1007/978-3-540-24769-2_10 · Source: DBLP

CITATIONS
9

READS
1,891

2 authors:

Rafael Capilla
King Juan Carlos University
149 PUBLICATIONS **2,171** CITATIONS

SEE PROFILE

Margarita Martínez
Escuela Técnica Superior de Ingeniería Informática
8 PUBLICATIONS **23** CITATIONS

SEE PROFILE

Software Architectures for Designing Virtual Reality Applications

Rafael Capilla and Margarita Martínez

Department of Informatics and Telematics,
Universidad Rey Juan Carlos, Madrid, Spain
{rcapilla,mmartinez}@escet.urjc.es

Abstract. Software architectures are particularly useful when designing complex systems. Apart from facilitating the design, development and evolution processes, software architectures help developers who are new in the domain to understand the design issues involved, reducing the learning effort. In this work we present a software architecture for virtual reality systems. This architecture applies patterns common in other interactive systems, such as the Model-View-Controller, and also identifies new patterns proper of the VR domain, such as the scene graph. In addition, in the proposed architecture we have identified the variability points needed for adapting and evolving such VR systems.

1 Introduction

The challenge of a virtual reality system (VR-system) is to simulate the real world in a virtual environment, making it real for the user who is immersed in the system [8]. Among the characteristics of virtual reality systems that make them complex to develop we can cite the following.

- Use of special hardware devices: head-mounted displays, 3D haptics, etc.
- Complexity of user interfaces and multimodal interaction.
- 3D modeling techniques.
- Complex graphic operations, such as object collision and deformation.
- Presence of the user in the virtual scene.
- Real-time requirements.

We believe that all these factors can preclude a quick development of virtual reality software applications. The challenge to facilitate the construction of such systems is the motivation of this paper. Therefore we have tried to provide some guidance for understanding and designing VR systems from a software engineering perspective. Our proposal analyses the use of *software architectures* [2] [4] [21] for building virtual reality applications in order to reduce the effort needed in the development process by making more modular and reusable its most relevant parts. The organization of the paper is as follows. Section 2 presents the related work in this field. Section 3 describes our approach for building software architectures for VR systems. Section 4 evaluates our work through the construction of a VR system and section 5 provides the conclusions obtained.

F. Oquendo et al. (Eds.): EWSA 2004, LNCS 3047, pp. 135–147, 2004.

2 Software Architectures in Virtual Reality Systems

We have found only a few proposals in the literature describing virtual reality systems employing software architectures. Schöntage and Eliëns [18] describe the problems for developing distributed VR systems and they mention four architectural styles for classifying object-oriented software under distributed environments. In [19] the DIVA Distributed Architecture Visualization is presented as an example of the use of architectural patterns [5] to achieve a software architecture [20] for building distributed VR systems. Another example is the Dragon system, which is a real-time battlefield visualization virtual environment [10]. The architecture is shown in figure 1.

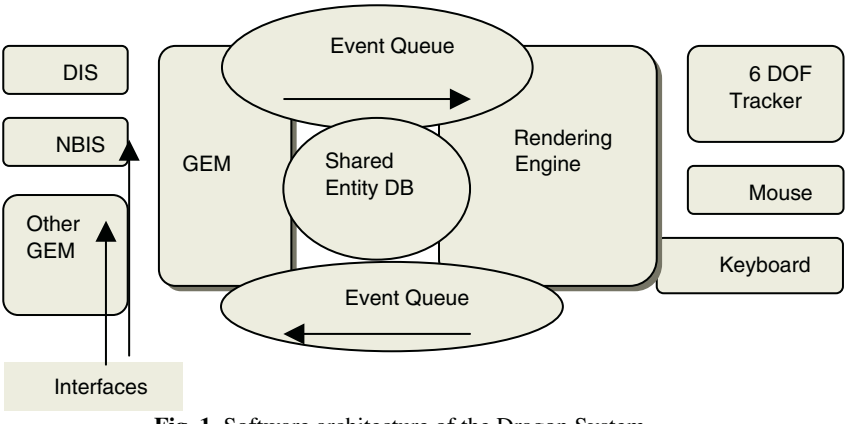

Fig. 1. Software architecture of the Dragon System.

The Dragon system is composed by two major subsystems: the *rendering engine* (RE) and a *generic entity manager* (GEM). RE draws the virtual environment, processes the user input and creates requests and events that are sent to the GEM subsystem. GEM is responsible to collect data from external sources and represent them under a common format. Both subsystems interact through a pair of unidirectional event queues and a shared entity database. DIS (Distributed Interactive Simulation) and NBIS (Joint Maritime Command Interaction) are interfaces of the system. In [3], the authors mention the requirements that software architectures must fulfil for supporting the design of VR systems as well as to perform rapid prototyping. Some of these requirements refer to the modularization and extensibility as a practical point of view in the design process.

In general terms, the lack of flexibility is a common point in many of the VR systems already developed. One of the problems in the development of Large Scale Virtual Environments [14] comes from the use of monolithic architectures, blocking some important aspects such as *maintenance, reusability* and *extensibility* among others. Therefore, a more modular design of VR applications based on good software architectures can improve the development and maintenance of these complex systems.

Other approaches that mention the use of software architectures in the construction of VR systems can be found in [1] [7] [9] but from our point of view, some of the references mentioned only show a global view of the system but not a more detailed one of the modules of the architecture. Moreover, the proposed architectures don't exhibit clearly the variation points needed for supporting the evolution of the systems we want to build. This is a key aspect for evolving the architecture through specific variability mechanisms as a way to avoid monolithic approaches. Finally, none of the proposals mentioned before describe the process by which they have obtained the architecture or which design techniques they have employed. In our opinion this is important if we need to develop other VR systems or evolve the existing ones as new requirements appear.

3 Architectural Construction Process

The process we have followed to obtain the software architecture is first analyzing the domain to obtain a domain model and build a reference architecture applying architectural styles and patterns. Finally we refine the reference architecture to obtain the software architecture for describing VR systems. These steps are detailed in the following sections.

3.1 Analysis of Virtual Reality Applications

A domain analysis process [17] is a key step for understanding the terminology and elements employed in most VR applications. Thus, the output of this process, that is a domain model, is used for representing those elements, characteristics and relationships that can be employed in the construction of the future software architecture. First, a d*omain scoping* process was performed to establish the limits of the domain and try to determine which VR systems will be inside or outside the domain. In this work we have included both immersive and non-immersive systems but we didn't consider distributed VR systems as well as those based on the CAVE (Cave Automatic Virtual Environment is a visualization device in which the images are projected in the floor and walls). The design of distributed VR applications is more complex because there are other factors such as communication cost, distribution of data and functions and coherence, while CAVE systems need a high cost hardware infrastructure not affordable for everybody. Finally, we didn't analyze VR engines because is not the main goal for VR application developers. Instead of this, we have preferred to analyze the most common types of VR applications that don't need stronger requirements. After this, in the *analysis phase* we identified the domain vocabulary extracted from several knowledge sources (e.g.: technical documentation, experts, web pages, technical guides, etc.), providing a classified list of terms. VR applications are implemented in several programming languages, so we had to compare similar concepts in order to classify them properly. Finally, we obtained several relationships between the

elements, its properties, actions and VR devices and other relevant parts of a VR application that we used us in the construction of the software architecture.

Before we produce the domain model we did a reverse engineering process from existing VR code in order to obtain additional information that leads us to a more accurate architecture. The code inspected belongs to small VR applications written in different languages: 5 VRML, 3 Java 3D, 4 C code with OpenGL and FreeVR functions and 2 Multigen's Vega examples. The process was done manually due to the diversity of code analyzed. The terminology may vary depending on the programming language used but elements of VR applications share similar concepts. For instance, a VRML [13] node can be associated to a Java 3D object. Some of the elements identified in this step were objects, properties, events, routes and behaviors. The identification of properties associated to the aspect of the object (e.g.: shape, color, texture, or size), served for the identification of the variation points in the architecture. Finally, we identified other relationships that were used to discover the structure of a VR application, such as figure 2 shows.

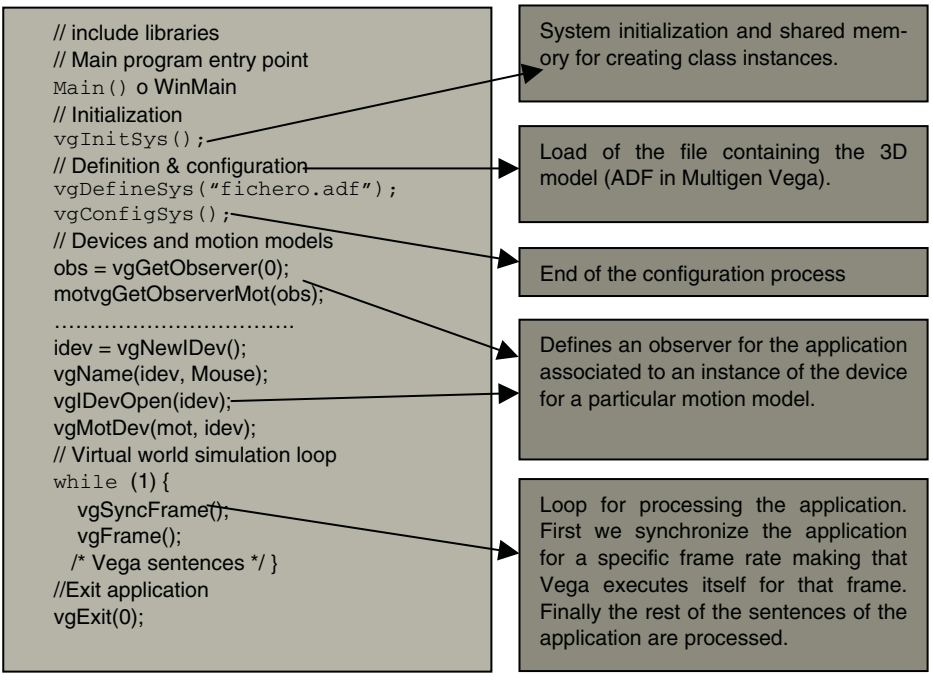

Fig. 2. Structure of a Multigen Vega Virtual Reality application.

3.2 Reference Architecture Development

Once the domain model was finished, we started the construction of the reference architecture. Reference architectures focus on fundamental abstractions for satisfying a set of reference requirements and constitute a previous step before we build the

software architecture, because they serve to understand the standard parts of a set of systems. The requirements needed for developing the reference architecture represent those needs that are usually associated to a whole domain for describing the particular characteristics of the problem space. For representing these requirements we used an object-oriented template from the IEEE Std. 830-1998 standard, such as figure 3 shows.

1. Specific requirements
    - 1.1 External interface requirements
        - 1.1.1 User interface
        - 1.1.2 Hardware interface
        - 1.1.3 Software interface
        - 1.1.4 Communications interface
    - 1.2 Classes/Objects
        - 1.2.1 Virtual scene: formed by 3D objects, avatars, lights and other objects.
        - 1.2.2 3D geometric objects
            - 1.2.2.1 Attributes: shape, color, size, etc.
            - 1.2.2.2 Functions: movement, rotation, etc.
            - 1.2.2.3 Messages: through events, routes.
    - 1.3 Performance requirements
    - 1.4 Design constraints
    - 1.5 Software system attributes
    - 1.6 Other requirements

Fig. 3. Partial list of reference requirements based on the standard IEEE Std. 830-1998.

We also evaluated some architectural styles that we used in the architectural development process. For instance, the model-view-controller (MVC) pattern is quite important in interactive systems and in VR systems results useful for representing multimodal interaction (i.e.: interaction through several devices and methods, such as pointing, gestures, force-feedback and more). The construction of the architecture usually takes several iterations before the final design is finished and in our first iteration we identified the following functional blocks.

1. *User interaction layer*: Comprises the user interface, which accepts the user input from several I/O VR devices and visualizes the output reflecting the changes performed in the virtual scene. This layer allows the operations specified in user interface.
2. *Information processing layer*: Constitutes the core of a VR system. The aim of this subsystem is to process the information given by the user input and enacts the operations and tasks specified in the application's requirements.
3. *VR engines layer*: One or more engines realize the operations for re-drawing the objects in the scene. For application developers, VR engines are not very significant because usually they are integrated under the development platform or runtime environment. Therefore we will consider this module as a black box that performs low-level operations for the functions specified in the information processing layer.

We represented the modules described above using a three-layered style [6] because the lower layers or modules provide the functionality needed by the higher ones.

In a second iteration we selected the model-view-controller for representing the higher and middle layers of the architecture because the MVC pattern allows a clear separation between the information being processed (i.e.: the model), the user input coming from different VR devices (i.e.: the controller) and the output (i.e.: the view) shown to the user. The *model* of the MVC style represents the information processing module (i.e.: middle layer of the architecture) whereas the *view* and the *controller* are placed in the higher layer (i.e.: user interaction layer). In some cases the virtual scene is visualized directly, but in other situations we can define several views. Examples of this situation are those VR systems that use the World-in-Miniature (WIM) technique [15] for navigating through the virtual world. A WIM model is a miniature 3D map that permits to the user move across the virtual environment. Finally, in the third iteration we completed the reference architecture specifying the UML classes and packages needed for representing the elements of each layer. For the higher layer we specified the controller and view classes and for the middle layer (i.e.: the model) we included the following ones:

- 3D object package: It represents the scene-graph composed by a hierarchy of 3D objects that form the virtual scene.
- Event package: This package describes the events needed by the objects for communication purposes.
- 3D sound class: Allows sound in the virtual scene.
- Lighting class: Permits the existence of light points in the scene. Several types of lights can be added and customized.
- Other 3D elements (class): To be defined for each particular application.
- Device initialization package: Device initialization can be done in this layer when this task is not supported by the higher level of the architecture.
- Specific packages needed by specific applications (e.g.: other graphic routines).

3.3 Software Architecture for VR Systems

Based on the reference architecture described in the previous section, the development of the software architecture was done by refining the packages and classes of the two higher layers of the design (i.e.: user interaction layer and information processing layer). The refinement process takes the requirements and features of VR applications and defines attributes and methods for each package and class that represents a particular functionality in the system. These attributes and methods are defined taking into account the variability of the future software architecture. For each VR device, software piece or 3D element of a VR application we have defined a set of customizable attributes and methods.

In this work we represent the static view [11] [12] of the system but other ones are possible if needed. In contrast to existing architectures of VR systems that don't provide customization mechanisms, we allow a customization process through specific

variation points [4] which are represented by attributes defined in the architecture and customizable parts of the methods. Therefore we can ease the maintenance and evolution against future changes. For instance, in table 1 the *DeviceType* attribute allows several VR devices (e.g.: 3D mouse, tracker, haptic devices or head-mounted displays) for multimodal interaction. This is particularly important in VR systems because one of the methods to achieve realism is providing natural interaction, simulating the ways in which we interact with real objects. Also, the initialization method may depend of the particular VR device employed. Another possibility is the specialization of the controller class by defining subclasses for different devices. On the other hand the *View* class draws and updates the views for a particular observer so we have included it as an attribute inside the view class.

Table 1. Packages, classes and variation points of the User Interaction Layer.

Package	**Class**	**Attributes (Variation Points)**	**Methods (Customizable parameters)**	**Description**
Subsystem. UI Layer	Controller	DeviceType	Initialization() ProcessEvent()	The controller class accepts the user input from VR hardware devices and processes the events generated by such devices. The controller and view classes are related in the MVC model and we have placed them n the same layer of the architecture. The information from the events processed are passed to the model (i.e.: information processing layer), so the VR application knows what to do when a new event arrives.
Subsystem. UI Layer	View	Observer	DrawView() UpdateView()	The view class draws or updates the views with the data generated by the model. Several views are allowed.

In the information processing layer we have refined the packages and classes already specified in the reference architecture, such as table 2 shows. For instance, we have defined two attributes (i.e.: two variation points) in the lighting class for detailing the type of light source (i.e.: local, infinite, etc.) and the position. Other properties for light sources such as attenuation or color can be also defined and customized afterwards. An interesting point in which there is not much work done from a software architecture point of view is in the definition of the scene-graph (i.e.: a tree representing the object hierarchy) of the virtual model. This is quite important in VR systems because the time needed for loading the virtual model may vary substantially depending on the structure of the object hierarchy (the typical size of a realistic 3D model is measured in Megabytes). The object class shown in table 2 holds properties such as size, color, shape or position, which can be customized.

The packages and classes given in tables 1 and 2 are organized in the software architecture shown in figure 4. As we mentioned in section 2, the existing proposals are usually poorly described and none of them employ UML for modeling the software

Table 2. Packages, classes and variation points of the Information Processing Layer.

Package	Class	Attributes (Variation Points)	Methods (Customizable parameters)	Description
3D objects	Root Group Object Polygons	For the object class(shape, color, texture)		This package contains a hierarchy of objects and elements for processing the 3D model stored in a database (i.e.: the scene-graph).
----	3D Lighting	TypeOfLight Position	Lighting()	One or more light points illuminate the virtual scene or particular objects.
----	3D Sound	TipeOfSound	Activation()	3D sound added to the scene.
----	Other 3D elements	-----	----	E.g.: such as fog or environmental effects.
	Player	MyObserver	MotionModel()	Sometimes we can add a player to an observer which has a specific motion model for animating the objects or avatars in the scene.
Events	Event Sensor	MyEvent MySensor	EventAction() Detection() EventSensor()	Events and information from sensors (e.g.: collision detection or VRML routes) can be manage in the virtual scene.
Specific packages	----	----	----	To be determined. Specific for a particular VR application or family of related systems.

architecture of a virtual reality system. This is important for understanding the key parts of a VR system in order to facilitate the construction of such systems by novice VR developers using standard design methods. From our point of view UML is suitable for representing most of the architectural decisions of VR systems. Other architectural views (i.e.: physical, process view) can be also represented employing other UML diagrams (e.g.: deployment, sequence diagrams, etc.). Perhaps the weakest feature of UML from our point of view is the lack of a way to represent in detail the variation points.

4 Customizing the Software Architecture for a Virtual Church

The evaluation of the architecture of figure 3 was done building a small VR system consisting in the design of a virtual church as well as a virtual tour inside the church. The church is a 12th century Romanesque temple located in a small village in the north of Spain (Cambre's village, http://www.cambre.org) and the shape is similar to old European cathedrals. The application consists in a virtual tour through the church showing its main characteristics and elements with comments that appear in key places. The user interacts with the system through a head mounted display, a tracker and a 3D mouse. For developing the system we used Multigen Paradigm software

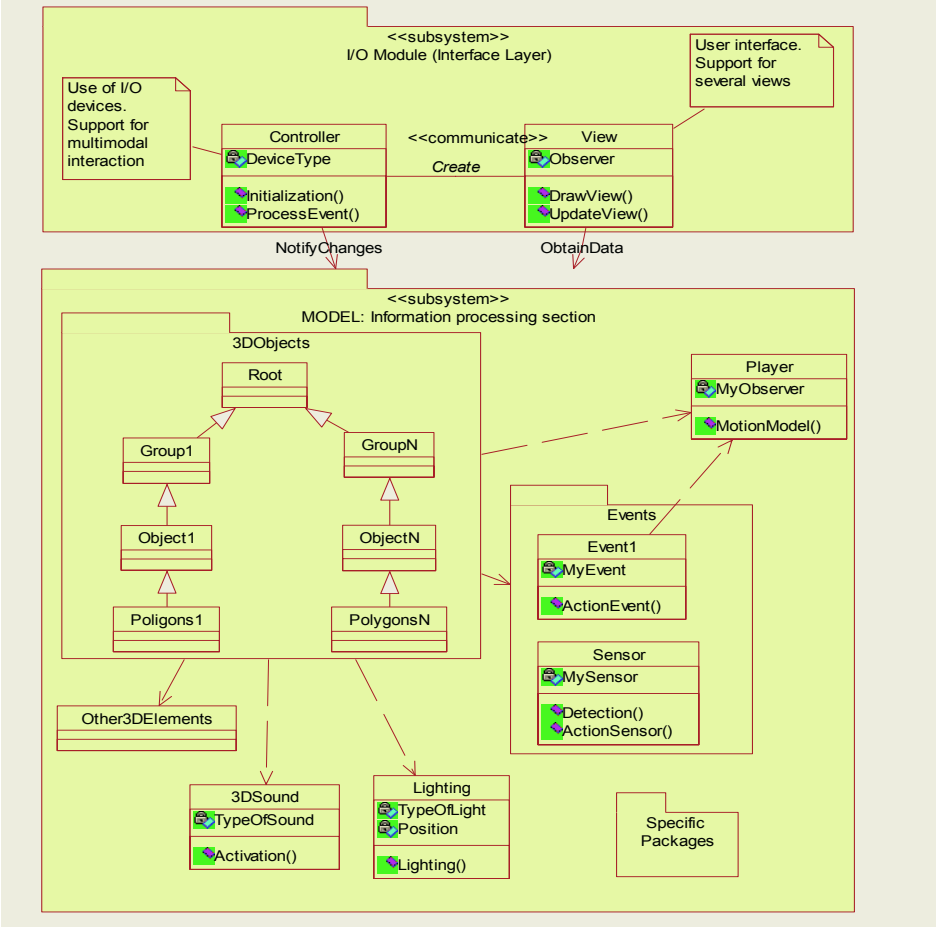

Fig. 4. Software architecture for VR systems.

tools. The application was developed by a novice software engineer not familiarized with the construction of VR systems. In order to facilitate the design of the system, we used the proposed architecture to explain to her the standard parts of the VR system. The customization of the architecture was done through the specific variation points already mentioned by filling the attributes with appropriate values. The requirements for the new application served as a guide for selecting which variation points should be included in the final architecture. Also, we used appropriate values for customizing specific methods that were reused for the final implementation (e.g.: initialization values for different hardware devices) and the final design was quickly obtained such as figure 5 shows.

Once the developer was familiarized with the Multigen Paradigm tools, we took the measures of the church and we started the construction of the 3D virtual model using the Multigen Creator tool. We customized the scene-graph of the 3D model and we decided which elements of the architecture would be used. The Creator database

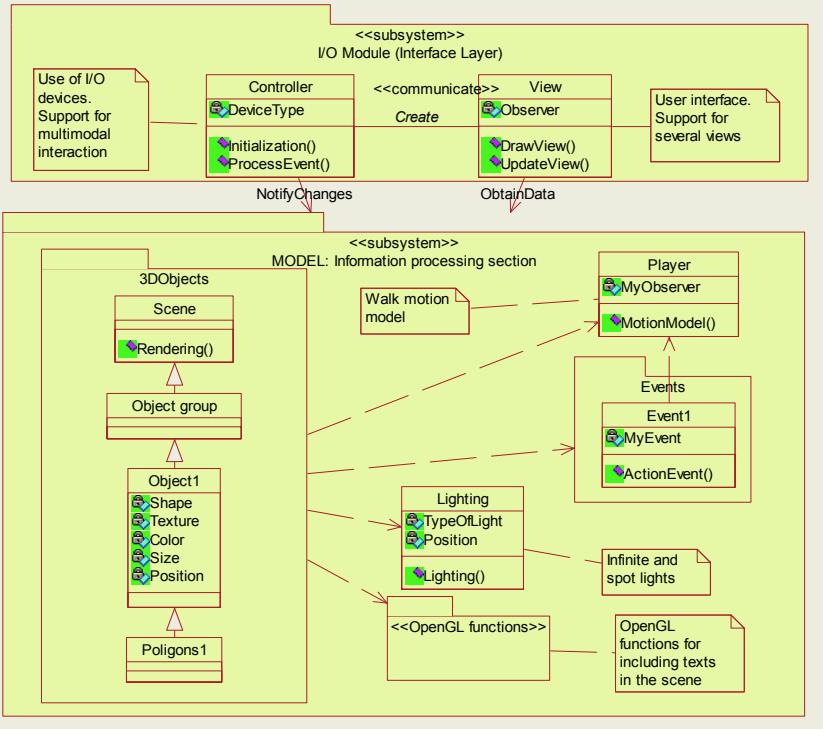

Fig. 5. Customized software architecture for a VR system.

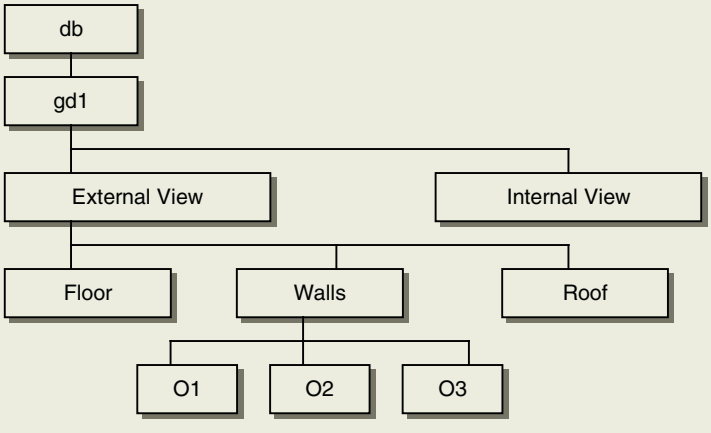

Fig. 6. Partial view of the object hierarchy.

organizes the scene graph starting with a root element called “db” and a generic group “gd1”, from which the rest of the objects hang on. We have divided the construction of the church in two main object groups: the external view and the internal one. For instance, the main three elements of the external view are the floor, the walls and the

roof. Several objects and polygons compose the "wall" object but the Creator tool assigns internal names for them (e.g.: O1, O2 and O3). Next, we added textures and colors for each object of the church in order to achieve a more realistic view. Figure 6 shows an example of this.

Other parameters of the customized software architecture are the type of motion model (e.g.: walk), environment effects (e.g.: lights) and the initial position of the observer. To do this we used the Multigen Lynx tool that facilitates the work of the designer, which generates an application definition file (ADF).

In this way, the variation points and the packages defined in the software architecture served as a quick guide for designers and developers to decide which elements should be in the final design. This makes more reusable such pieces of code so they can be quickly incorporated to the definitive implementation. The way by which we customize the software architecture depends on the application and on the specific VR tools employed. In our case we used the Multigen Creator and Lynx tools but in other cases this process may be managed in a different way.

At last we used the Multigen Vega platform for programming the details of the virtual tour and we produced an executable file written in C with calls to Vega and OpenGL functions. We added specific programming for processing events coming from the I/O devices. The application receives events from the 3D mouse to stop or initiate the movement of the observer, and from the tracker to detect the position of the user head and modify the view accordingly. A predefined path was specified to perform the virtual tour. Also, at specific interesting points of the guided tour we added some textual information that allows the user to learn the history, characteristics and elements of the church. This feature was performed through OpenGL functions, also reflected in the architecture as a specific or a domain package.

5 Conclusions

In this work we have proposed a new software architecture for VR applications that can reduce the development effort, particularly to people not familiarized with this kind of applications, by providing a good understanding of the elements of a VR system and their relationships. Compared to other existing proposals, we have outlined more accurately the standard parts and functional blocks and we have represented them using the UML notation. Moreover, the definition of specific variation points in the architecture helps designers to decide where the software architecture should be customized for a specific application and facilitates the maintenance and evolution of VR applications over time. The construction of a VR application can be performed more quickly because the standard parts of a VR system have been designed more reusable compared to existing proposals.

Related to this, the modification of the variation points has an immediate effect in the architecture that can be evaluated both at the design of the 3D model as well as viewing what the user sees or feels when interacts with the VR application through specific VR devices. The simulation of the design with a real implementation is the best way to test the proposed architecture.

Another result we obtained refers to the multimodal interaction of VR systems and the use of several hardware devices. This topic has been gathered by the architecture with the controller and view classes of the MVC pattern. The customization of the controller class allows the use of different devices used by modern VR systems and implements different hardware initialization processes if needed. Also, the architecture can support several views if complex graphical user interfaces are needed. For instance, an application could use one view for representing the general layout of a historical place and a detailed view for each for the most interesting parts of the virtual model selected by the user.

Other interesting aspect is that the architecture explicitly represents the scene-graph of the 3D model. This is an important issue in the design and execution of VR systems because many times the performance of the system depends on the organization of the objects in the scene-graph. For the future, we will like to extend our architecture to include other VR applications (e.g.: distributed ones or CAVE-based systems) and test the suitability of our architecture for supporting the evolution against future changes. In addition to this, exploring other scene-graph architectures based on software patterns is an interesting direction for research in VR systems. Finally, quality attributes such as performance, usability and presence are important to be explored for validating non-functional properties in the proposed architecture. Measuring the values obtained through the execution of the system is a way to provide results when defining such quality attributes.

References

1. Alexandre, R.J. F. and Medioni G. G. A Modular Software Architecture for Real-Time Video Processing, Proceedings of the 2nd International Wokshop on Computer Vision Systems, 35-49 (2001).
2. Bass, L., Clements, P., Kazman. Software Architecture in Practice, Addison-Wesley, 2nd Edition (2003).
3. Blach, R., Landauer, J., Rösch A. and Simon, A. A Highly Flexible Virtual Reality System. Future Generation Computer Systems Special Issue on Virtual Environments, Elsevier, Amsterdam (1998).
4. Bosch, J., Design & Use of Software Architectures, Addison -Wesley (2000).
5. Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P., Stal, M. Pattern-Oriented Software Architecture. A System of Patterns. John Wiley & Sons, New York (1996).
6. Clements, P., Bachman, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Stafford, J. Documenting Software Architectures, Addison-Wesley (2003).
7. Fernando, T., Murray, N., Tan K. and Wimalaratne, P. Software Architecture for Constraint-based Virtual Environment, ACM International Symposium on Virtual Reality Software and Technology (VRST'99), London, UK (1999).
8. Gobbetti, E. and Scaneti, R. Virtual Reality, Past Present and Future, Online at: http://www.csr4.it/vvr/bib/papers/vr-report98.pdf
9. Hua, H., Brown, L. D., Gao C. and Ahuja, N. A New Collaborative Infrastructure: SCAPE. IEEE Virtual Reality (VR'03), (2003).

10. Julier, S., King, R., Colbert, B., Durbin J. and Rosenblum, L. The Software Architecture of a Real-Time Battlefield Visualization Virtual Environment, IEEE Virtual reality, Houston, Texas, USA, (1999).
11. Kobryn, C. Applied Software Architecture, Addison-Wesley (2000).
12. Kruchten, P. Architectural Blueprints. The 4+1View Model of Software architecture, IEEE Software, 42-50, (1995).
13. Nadeau, D. R. Building Virtual Worlds with VRML, IEEE Computer Graphics and Applications, 18-29, IEEE Computer Society, (1999).
14. Oliveira, M., Crowcroft, J., Slater M. and Brutzman, D. Components for Distributed Virtual Environments (VRST'99), 176-177, London, IK, (1999).
15. Pausch, R. and Burnette, T. Navigation and Locomotion in Virtual Worlds via Flight into Hand-Held Miniatures, SIGGRAPH'95, ACM, 399-400, (1995).
16. Poupyrev, I. and Ichikawa, T. Manipulating Objects in Virtual Worlds: Categorization and Empirical Evaluation of Interaction Techniques, Journal of Visual Languages and Computing, vol 10, 1, 19-35, (1999).
17. Schäfer, W., Prieto-Díaz R. and Matsumoto, M. Software Reusability, Ellis Horwood, (1994).
18. Schönhage B. and Eliëns, A. From Distributed Object Features to Architectural Styles, Workshop on Engineering Distributed Objects (EDO'99), International Conference on Software Engineering (ICSE), Los Angeles (USA), (1999).
19. Schönhage, B., van Ballegooij, A. and Eliëns, A. 3D Gadgets for Business Process Visualization, International Conference on the Virtual reality Modeling Language and Web 3D Technologies, Monterrey, California, USA, (2000).
20. Schönhage, B. and Eliëns, A. Information Exchange in a Distributed Visualization Architecture: the Shared Concept Space Proceedings of Distributed Objects Applications (DOA'00), Antwerp Belgium, (2000).
21. Shaw, M. and Garlan, D. Software Architecture. Prentice Hall (1996).