2014 IEEE 17th International Symposium on Object/Component-Oriented Real-Time Distributed Computing

Modular distributed visualization and collaboration for a real-time 3D visualizer

Alexandre Valdetaro*, Alberto Raposo* and Pablo Elias*
*Technical-Scientific Software Development Institute (Tecgraf)
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil
Email: xvaldetaro@gmail.com, abraposo@tecgraf.com.br, pelias@tecgraf.puc-rio.br

***Abstract*—In this work, we present the design and implementation of distributed visualization and collaboration for an immersive 3D visualizer in a component-oriented fashion. The design follows an MVC approach, isolating all the business objects in the lowest level of the application, making it modular and extensible, therefore providing an easier prototyping of functionality and the isolation of complex business logic algorithms. This design as a solution came from the necessity of an existing visualizer with a monolithic implementation, whose maintainability and improvement are impaired due to the coupling between the business logic and the diverse visualization and distribution algorithms. Our design may be reused as an inspiration for other visualizers that wish to reduce the complexity and cost of the development of new business functionality. We implemented the designed visualizer, where we verified both the synchronism of the distributed visualization and the consistency of the collaboration among multiple nodes. We also evaluated the performance impact caused by the distributed visualization.**

I. Introduction

Immersive applications seek to provide an experience to the user as close to real life as possible. Such experience must re-create sensorial stimuli and perception of space in order to induce the user's brain into believing that the immersive experience is real life. Although an immersive experience can be delivered by any kind of system, Real-Time 3D visualizers may stand out as the key player in the field. A visualizer is an application that enables the user to visually explore a virtual scene. The success of such applications is consequence of many features combined, such as, credible visual input to the user by usage of modern computer graphics techniques, enhanced illusion of visual depth due to stereoscopy, precise and immediate control thanks to real-time reading and processing of user input. Therefore, the development of an application that provides an immersive experience to the user can be very complex because of these many requirements and their implementation in a real-time reactive system.

A. Distributed Visualization

Distributed visualization is the process of displaying a single virtual scene from multiple views. Such process faces the technical problem of how to render and synchronize so many views and output to different screens. A traditional approach to achieve distributed visualization is through the usage of out-of-the box systems that make use of dedicated hardware. The hardware controls all the visualization devices. Such solution presents, usually, a high cost of deployment and maintenance.

The rendering speed depends roughly on the amount of geometry—the number of vertices of every object in the viewable scene—being processed and the size of the output image. Therefore, rendering multiple views of the same scene gets very expensive in terms of processing, specially if the output images have to be very big for display walls. Consequently, distributing the scene to multiple nodes can be a feasible solution—despite the distribution difficulties. There are many possible designs of a visualizer that provide scene distribution as well as many tools available for it. In the related work section we make a review of the main available tools and their usefulness for our scenario.

B. Collaboration

Collaborative software is a software that enables multiple users to collaborate to achieve a common goal [1]. A collaborative visualizer lets the users explore a common scene, and possibly make changes and view other users avatars in the scene. The most common case of a collaborative visualizer is a multi-player game, where the players interact with each other in a shared virtual world. The players can talk to each other, engage non player characters and change the persistent world in many ways.

C. Real-time Visualizer

Any real-time application must be reactive, i.e., it must process input from external devices with a delay small enough not to break user immersion. Moreover, the interactive visualization requires that at least a given amount of frames be rendered every second. Thus, efficiency is the utmost requirement when developing a feature in a real time 3D visualizer.

In order to achieve maximum efficiency in a visualizer's many routines, e.g., rendering, input processing, distribution, data loading and so on, there is a natural tendency of the developers to trade abstractions for low-level APIs in order to have access to every available optimization setting. However, the exposure of every low level API to the business logic developer can greatly increase the complexity, which may lead to an increase in the lifecycle cost [2] and effort [3] required to maintain and extend a visualizer with new functionalities.

To the best of our knowledge, the majority of the 3D visualizers, where efficiency is paramount, are implemented in a monolithic way. There is no clear separation between business, distribution, rendering, architectural elements and

1555-0885/14 $31.00 © 2014 IEEE
DOI 10.1109/ISORC.2014.44

IEEE computer society

so on. This approach can have some advantages because the development of every functionality has access to every low level system, therefore the developer can highly optimize the rendering, distribution and other techniques used in the developed feature. However, from our experience (see section II), the constant increase in complexity for the development of simple business/application features, and the tight coupling between the logic layers can seriously impair the maintainability and extensibility of the application. This can lead to a stall in productivity, where developers suffer with the high complexity they need to understand in order to develop a simple feature.

We believe that the application programmer, who develops the features that aggregate real knowledge value in a software, should be focused on the domain of the application, the business logic, transactions, user interface and any other front end concern. Therefore the application programmer should work with as much high level abstractions as possible, such as frameworks, tools and *middleware* [4]. However, one must always consider the trade-offs when abstracting low level systems, as the usage of low level optimizations may be necessary in many cases as explained above

This work is organized as follows: In section II we show our background and motivation for this work. In section III we walk through the available solutions for our distributed visualization and collaboration requirements for an immersive visualizer. In section IV we explain our design for a visualizer oriented towards modularity and extensibility. In section V we explain our design for the module that provides distributed visualization and collaboration for the visualizer. In section VI we show the performance of the implemented example system. In section VII we present our conclusions and thoughts about this work and in the last section we indicate some ideas for future work.

II. Background and Motivation

This work is developed as a part of SiVIEP (Integrated Visualization System of Exploration and Production) project. SiVIEP is an immersive scientific visualizer, which supports visualization of reservoirs, geological surfaces, wells, risers, platforms and many other objects from the oil field domain. All of these objects can be visualized together inside one project in a 3D multi-screen stereoscopic setup, controlled by several manipulators and tracking devices. Many supported objects pose already a challenge to be rendered in a single screen due to the number of polygons, simulation data and property visualization. Therefore, complex distribution algorithms must be employed for the system to be able to render all the required objects in a multi-screen stereoscopic setup.

SiVIEP's current production version is a monolithic *C++* application using Qt for user interface and OpenSG [5] for distributed rendering. There are some bottlenecks with the distribution provided by OpenSG [5] for our scenario, and these are discussed in the related work section. Also, the complexity of adding new business objects to the current *C++* version is too high due to the tight coupling between many non cohesive parts of the application.

We started this work to provide a solution for these SiVIEP issues with a new design that could isolate the rendering and distribution complexity from the business logic.

III. Related Work

A number of generic solutions for distributed visualization have been developed, and are available commercially or open source, e.g., Equalizer [6], Chromium [7], [5] and VRJuggler [8]. Some of those are completely transparent, some require a certain level of adaptation from the programmers and some force its own programming model. In the collaboration field, most of the state of art solutions are far more complex and generalist than desired for this work. However, some of these works contributed to model our solution and are going to be detailed here.

OpenSG [5] is a scene graph as much as any non-distributed scene graph. A user can develop using OpenSG without distributed visualization in mind despite being one of the scene graph's major features. The distribution strategy of OpenSG works by distributing the scene graph with all the graphical nodes containing vertices, textures, matrices and other graphical primitives. Therefore all the graphical nodes must be serializable. Making a serializable graphical-node is simple if it is a regular triangle mesh that requires no processing during runtime because the node will be serialized once during pre-processing only. However, many nodes have a constantly varying set of primitives such as: large on-demand loaded files such as terrains, photo-realistic detailed meshes with continuous level-of-detail techniques, simulation data visualization. These variable nodes are very expensive to distribute, as their data is constantly varying and can be of very large size. Taking our oil field visualizer scenario as an example, most of our domain is composed of simulation data, that requires different visualization modes with generated graphical data on demand. Therefore, this graphical distribution approach is not suitable for us.

Chromium [7] works by creating a powerful abstraction of the OpenGL API, that is, it intercepts every call to the API and executes a scalable rendering algorithm to distribute the call among multiple nodes. Such approach is very clean and non-intrusive, but requires the constant distribution of graphical data, which can be a bottleneck if the scene is very diverse so that the GPU memory will need to be updated frequently. Also, this approach does not consider collaboration. In order to provide collaboration with Chromium, a whole separate solution must be implemented.

Equalizer [6], Chromium [7] and VRJuggler [8] are tailored for parallel and scalable rendering, that is, using multiple nodes for rendering the same scene independently of the number of screens. They all support outputting the rendered scene to multiple screens making distributed visualization feasible. One of these tools stands out as the most complete and non-invasive, the Equalizer [6]. It is a very powerful and complete tool that forces the programmer to abstract the rendering code from the rest of the application. Thus, it distributes this rendering client for the slave nodes. Such nodes perform rendering tasks controlled by a master node, which is configured by configuration files describing the available resources in the cluster as well as the desired compositing strategies. The application is unchanged for any kind of scalable setup. The rendering in the slave nodes can be configured for many different compositing strategies [9], such as sort-first or tile based, DB based or sort-last, among others [10]. After one of these strategies is used, the output image is copied to the configured output walls/screens. Such strategy of scalable rendering is very powerful, but unnecessary for a typical case of multi-screen rendering setup (our whole range of cluster examples fit the common case) where every screen is driven by an individual node of the cluster. Also, if the application is going to be used in single computer workstations to visualize the same projects, then using the scalable rendering system to achieve better system results can be pointless, as the scene needs to be processed by a single machine as well.

All of the aforementioned solutions provide the distribution on the graphical layer of the application, that is, they distribute graphical code, graphic controlling commands, graphical primitives and so on. However, for a collaborative visualization to be achieved, a distribution of the business domain (not just its graphical representation) is necessary. Therefore, in the mentioned approaches, the collaborative visualization would have to be solved by other means. In this work's solution, we provide a DSO (distributed shared objects) system for our domain data. That is, our domain data is a single shared graph among the nodes. This way, our solution provide collaboration through two-way updates of local changes for every node. Moreover, this same solution will serve as the ground for our multi-screen rendering algorithm to work with, as all the graphical representation can be loaded locally based on the received domain information.

IV. Design of a Modular Visualizer

A. Domains and Entities

The domain of an application represents *business knowledge*—as the *model*—in a *Model View Controller*—MVC—paradigm. What we have referenced in the previous sections as *business data* or the *business objects layer*, is technically the *application domain*. It is a mirror of the real world, composed of all the real world objects that the application wants to represent. Ideally, there should be a one-to-one correspondence between the domain objects and their real world counterparts. These domain objects are represented by *Entities*. Examples of entities in an oil field visualizer application are risers, wells, reservoirs and so on. We designed the domain structure as a graph and the entities as collections of attributes with no relational structure associated. Furthermore the graph and its entities can be versioned, shared and persisted if needed.

B. Scenes

We define a *scene* as a visualizable collection of entities with behaviors and tags. This definition however, may be considered incorrect because of our definition of domain, as the scene clearly has different properties and elements than a real world object. Moreover a more purist definition could consider the domain itself a scene, which is the case in many visualizers. Nevertheless we regard the scene as business knowledge because they represent snapshots of the same domain in different times, configurations, simulation scenarios and so on. Figure 1 shows the structuring of our domain regarding scenes. Examples of scenes in an oil field visualizer are: the oil field when exploration began, the oil field in present date, the oil field in a catastrophe simulation scenario, etc .

Figure 2 shows the layering of the system according to the MVC concept and an overview of the interfaces and interactions among the many modules.

C. Domain Model Framework

As mentioned above, the domain should be an object graph. However, this graph is not useful for the rest of the application only by itself. Its state needs to be propagated through the layers above it. Therefore it is necessary to have a graph controlling system that provides controlled access to the domain graph and notifications to everyone that depends on it. We designed this controlling system as a non-intrusive module that keeps track of all the changes in the object graph and provides a fine grained notification system for the rest of the application. We call this system the DMF—Domain Model Framework—(Figure 3). Every component that applies some modification in the business object graph needs to certify that these changes will be propagated to everyone else. Hence whenever a change has to be applied to the graph, this change has to go through the DMF. Consequently, every component that needs to display information about a business object needs to be connected to the notification system of the DMF.

The DMF notification system follows the *Observer* pattern [11]. Every element that needs to watch the state of a piece of the domain will be registered as an observer to that piece in the DMF notification system. Accordingly, the DMF will notify that observing element whenever a change is applied to the observed piece of the domain. The notification itself consists of the previous and the current states of the piece. The DMF also provides a global graph access system for the elements that need to change the graph. Notice that in our design, the actual objects from the business object graph does not reside inside the DMF data structures. As we mentioned before, the domain management is non-intrusive. Therefore, the DMF has an internal graph containing meta information about the actual objects and it keeps versioning information, which is necessary for undo/redo operations and is also very important for our distribution module that will be explained in the next section.

This non-intrusive approach makes the DMF useful for any graph of objects that must be versioned and shared. However, as the actual objects are not inside the DMF, there is the risk of a business object being modified from outside of the DMF interface. If this happens, then there will be an inconsistent state. Therefore, there must be strict guidelines for the developers on the usage of such kind of framework.

With these foundations set, a module responsible for displaying the 3D scene can be a simple observer of the current scene and its referred entities, which are all contained by the domain graph. Whenever a new entity reference is added to the current scene, the module searches for a graphical representation for the object and add it to the underlying graphical API. Never should the actor be modified directly, it must always mirror the business object through observation. This way, it becomes easy to switch the graphical implementation of the objects. We may change the whole scene graph implementation without touching the rest of the application.

The UI layer sits above the rest of the application, no module should access the UI. The UI must make sure also

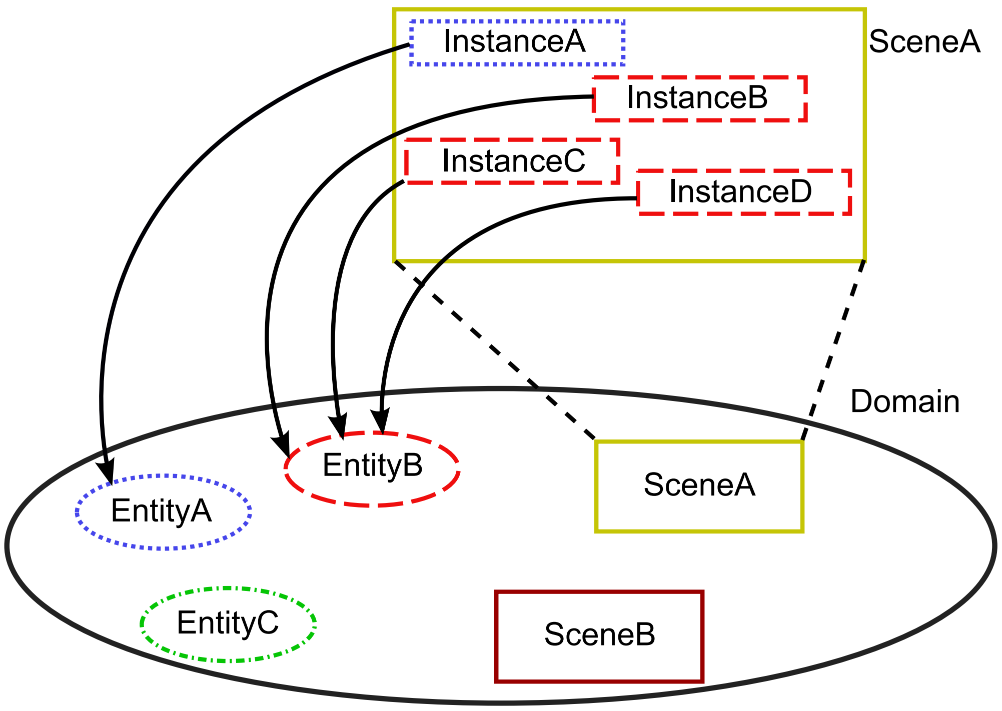

Fig. 1. Chart showing the relation between the domain, the scenes and the entities

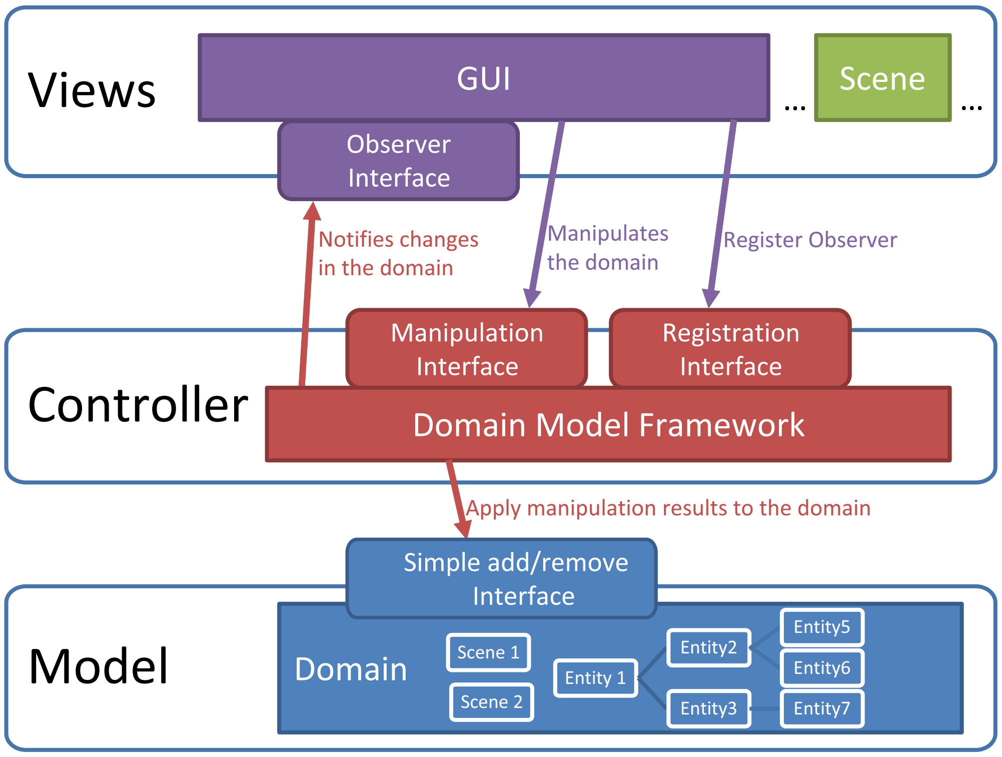

Fig. 2. Blueprint of the the visualizer architecture

that it does not change its own state, which is very common if no special care is taken. E.g., if a button that when pressed automatically changes its image and the button represents a state of a business object, when the state of the object is changed by any other means—be it collaboration, animation, task scheduling and so on—the button will be left in an

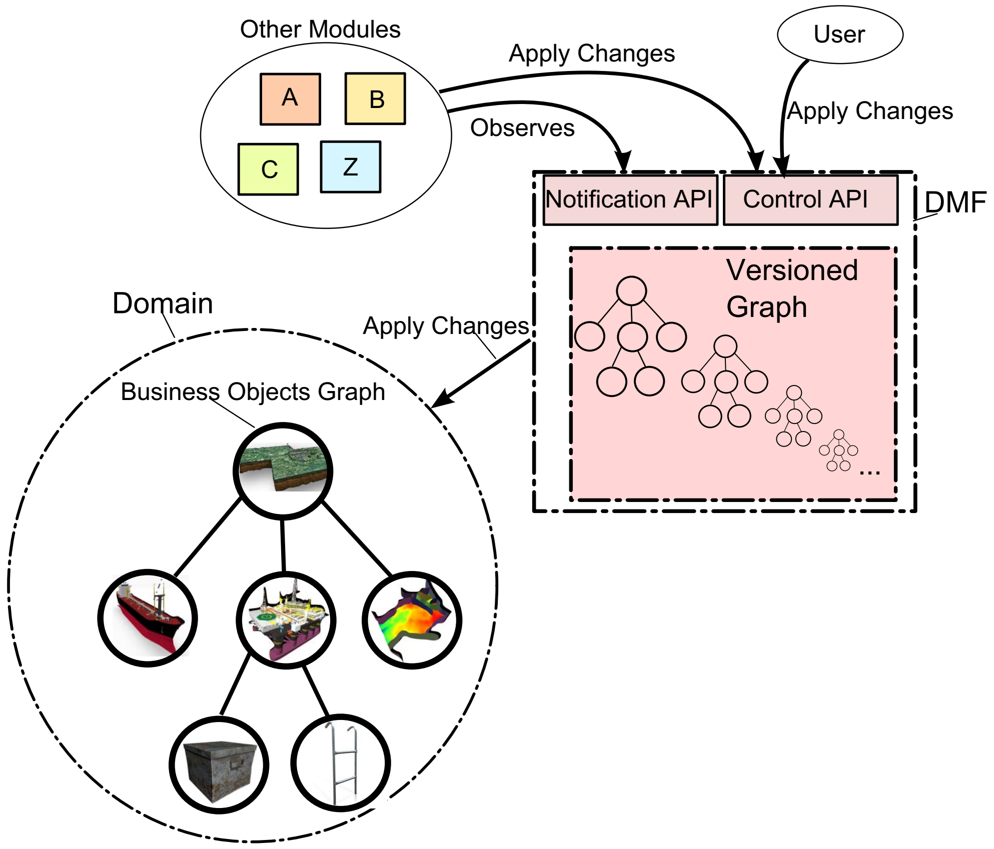

Fig. 3. DMF architecture overview

inconsistent state. Therefore, the developers must make sure that these automatic feedbacks that usual UI frameworks have by default are not enabled.

V. Distributed Visualization and Collaboration Module

A. Domain Distribution

We intend to seamlessly connect this module to our visualizer, hence transforming its visualization and workflow with only small changes in configuration files. Nevertheless switching from local to distributed may create a high risk for inconsistency if the application is not properly designed. This inconsistency can be noticed when the application state is spread across multiple elements and layers, hence making complicated for one instance of the application to propagate its state consistently to another. However, in our designed visualizer, we concentrate all the application state in a single element—the *domain*—thus alleviating our effort designing distribution. Still, in order to provide seamless integration with the application, this module must distribute the application domain in an agnostic and non-intrusive way.

We followed the same approach as with the DMF design—explained in Section IV-C—, we created a separate module with its own internal structures that serve only for the module purposes, the actual application domain data is controlled by the application. Hence the module *observes* and *applies* state changes to the application domain, for that purpose it has a dependency to the DMF. Therefore, our domain propagation strategy is clearly a mere distribution of the DMF notification calls, and persisted through the DMF domain control functionality.

B. Collaboration

In our shared domain scheme, whenever multiple users are making a collaborative visualization, they can simultaneously change the same data, therefore collisions are prone to happen. In online gaming realm, collisions are a very difficult problem to handle because there can be a massive amount of players interacting with the same data at the same time. Moreover there is usually no action of “take control of an entity” before altering it in any way, as the players are interacting with a simulated world, and requiring such action may break the immersion. Therefore, the collision problem needs to be handled very fast and smoothly so that all the players involved in the collision feel that the object that triggered the collision is being really shared. Such steep requirements are not the case in the scientific visualizer.

Entities in a scientific visualizer domain are not to be treated as objects in a game. It is possible to apply a system where a user must first take control of an entity in order to apply any changes to it. Such system eliminates the problem of users sharing an object. Still, even if there is no control system, the latency when solving a collision does not need

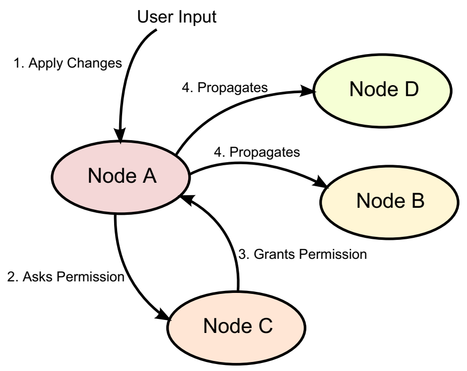

Fig. 4. Arbiter Topology

to be so low as in games. Therefore, it is acceptable to have an arbiter node responsible to keep the consistency, and every change applied by every node must be verified by the arbiter first, and only the arbiter and other designated nodes by the arbiter can propagate the changes (see Figure 4).

C. Distributed Visualization

Considering that all the domain data is properly distributed as explained in the previous section, achieving distributed visualization becomes a simpler issue. We need to calculate the views for every display output based on a single observer and synchronize the frame displaying. These tasks must be assigned somehow to nodes in the visualization cluster, hence we need to create roles with appropriate responsibilities for the nodes. In order to accomplish distributed visualization in our design, it is imperative that we assign at least master and slave roles.

The frustum calculation is done based on the output screens. Every node has a number of outputs connected to it and they need to display the current view of the scene on those. The view of the scene is based on a single observer—in a typical scenario—and therefore, this view should be split between the nodes. The master node needs to know the dimensions of the multiple-display environment and slave nodes disposition.

The frame synchronization is very simple. The master node must first command every node to render—with the appropriate frusta transferred. Then, after *every* slave node has finished rendering, the master node commands the slaves to present their rendered frames. If this synchronization strategy is not used, the user immersion can be broken, as there will be different frames being exhibited at the same time.

D. Integrating with the Visualizer

The Distributed Visualization and Collaboration Module—*DVCM*— should be a generic DMF observer that propagates local changes to the network and network changes locally. In order to use the module, when assembling the visualizer's components through a startup script, there must be a list of available options in the *DVCM* interface in order to assign its role. These options must cover a series of supported roles: local domain updates to be propagated throughout the network, incoming domain changes are going to be analyzed for collisions and confirmed or simply accepted, passively or actively render and display on the screen, among others. All of these possible setups are going to be covered and shown implementation-wise in the following section.

VI. Results

We show in this section how the implemented example system behaves under local and distributed visualization. Our goal is to measure how much the distribution and frame display checkpoint impacts the performance. Therefore we registered the frame rates of the visualization while varying the number of nodes in the cluster and which nodes used. As mentioned before, the example system has no multicast implemented, thus all the tests have been executed with only unicast communication between the nodes. We used 4 nodes with similar configuration in our tests:

Our tests are consisted of the visualization of a scene with 1,300,000 triangles with all culling disabled. This scene is just a collection of triangles in a certain position, but emulates a typical scene in our application usage scenario. During this visualization, we recorded the frames per second while the camera followed a predefined path. We executed these tests for a series of combination of nodes as shown in table.

In table I, we show in the ”Average FPS“ row how many frames per second each setup achieved and in the ”FPS loss“ row the loss of performance of the given node group when compared to the performance of the worst local node performance of the group. Notice that the first 4 scenarios are of the nodes executing the tests locally. The results show that there is a performance loss of less than 15% with networking due to latency of the network in any of the cases. Moreover, considering that nodes C and D have a similar performance locally and since a group performance limit is equal the weakest node's local performance, the comparison of performance between groups A,B,C and A,B,C,D is important to show us the raw impact of adding another node to a group, which is a negligible loss of less than 1%.

As can be seen, although the performance suffers an expected loss with distribution, our system can easily provide interactive frame rates during the visualization of large scenes in a distributed visualization scenario I, which is the usual desired cave disposition and our ultimate goal with the project.

We tested the system in the same 4 node scenario with a collaborative scene following an arbiter topology. The tests successfully worked and there had been no inconsistencies or noticeable performance hindrances as expected.

VII. Conclusion

In this work, we presented our design of an extensible and modular visualizer as well as the design of a module that provides distributed visualization and collaboration for the visualizer. By following these designs, we implemented an example system and tested the synchronism of the distributed visualization and the consistency of the collaboration among multiple nodes, we also evaluated the impact on performance caused by the distributed visualization.

Group	A	B	C	D	A, B	A,B,C	A,C,D	A,B,C,D
Average FPS	430	416	220	216	356	199	195	191
FPS loss	—	—	—	—	14.4%	11.3%	11.5%	11.4%

TABLE I. FRAME RATES AND PERFORMANCE COMPARISON OF DIFFERENT CLUSTER CONFIGURATIONS

We discussed first the relevant concepts and problems of designing and implementing a real time immersive visualizer, from which we extracted our main architectural requirements—modularity and extensibility. Following we presented our design for the visualizer inspired by MVC architecture and explained how we try to fulfill the aforementioned requirements. We continued by presenting the design of the module responsible for distributed visualization and collaboration, its interactions with the visualizer and why our described design for the visualizer simplifies its implementation and usage. We concluded with results of the implemented example system.

We expected to indicate how our MVC-inspired design made possible the development of a module that transparently provided distributed visualization and collaboration to a visualizer. Also, how the design enabled the substitution of parts of the system easily. E.g., the scene graph implementation can be switched between a very efficient and licensed per station library for displaying a scene in massive immersive environments and a cheaper licensed library for common desktop usage. These reasons along with others explained in section I are what motivated this work as a solution for our real project. We try to summarize here the key points that we believe we have addressed with our design and can impact overall productivity of the development of real time and efficiency-focused applications.

We addressed complexity with the separation of a system into modules with explicit interfaces, which tend to isolate the low level details of implemented features and create abstractions, which enhance the productivity of the business logic development by reducing the complexity of the system. The isolation of all the distribution code inside our designed module leverages the productivity of the business and graphical developers in our visualizer.

We provided easier prototyping for a product with a long development cycle, which can be extremely complex. Therefore, designing and implementing the system iteratively can leverage productivity and also accommodate late changes in product requirements. By making the application extensible, we give the developers an easy way to prototype new functionality and implementations. Our component based design enabled our system to be distributed after its first version without any significant modifications to it.

We enhanced the flexibility of the system with the possibility of switching between different functionality with no code change, which is important for a software that must assume different roles depending on the scenario—e.g. different scene graphs, distribution schemes and so on.

Our example system implementation showed that our design worked for the expect scenario. The synchronism among multiple output displays was achieved without any dedicated hardware and also without even the need to use broadcast messages. A loss of less than 15% for a 4 node setup, which is our common CAVE scenario has been a very good result, confirming our design as a proper solution. Furthermore, the consistency among the scenes when using the software collaboratively was achieved without any noticeable performance hindrances.

VIII. FUTURE WORK

Our distributed visualization strategy is not suitable for scenarios where there is no single node in the cluster with a number of output displays that creates a rendering bottleneck on it, which happens due to the amount of graphical processing on it. However, if desired, the task of rendering can be separated for the task of displaying. By creating this new layer of parallelism, the rendering task can be distributed equally among the nodes independently of the number of output displays connected to each one of them. However, it becomes necessary to recompose the final image based on a given recompositing strategy, which can adds complexity and inefficiency if not necessary.

The consistency among multiple stations when working collaboratively is currently only designed for reliable environments without simultaneous edition of the same entity. Therefore no special collision treatment of time synchronizing strategy is currently needed. However, if the visualizer needs to be deployed in a slow and non reliable network, with dynamic entity behavior, simultaneous edition of the same entity, and so on, the distribution module can be extended to support prediction algorithms, time synchronizing strategies and dynamic behavior descriptions.

ACKNOWLEDGMENT

This research project is supported by Petrobras. Alberto Raposo also thanks CNPq for the individual grant.

REFERENCES

[1] C. A. ELLIS, S. J. D. GIBBS, and G. REIN, “Groupware: some issues and experiences,” *Commun. ACM*, vol. 34, no. 1, 1991.

[2] B. Boehm, *Software Engineering Economics*. Prentice Hall, 1991.

[3] Microsoft Patterns and Practices Team, *Microsoft Application Architecture Guide*. Microsoft Press, 2009.

[4] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice.* Wiley Publishing, 2009.

[5] “Opensg,” www.opensg.org.

[6] S. Eilemann, M. Makhinya, and R. Pajarola, “Equalizer: A scalable parallel rendering framework,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, pp. 436–452, 2009.

[7] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski, “Chromium: a stream-processing framework for interactive rendering on clusters,” *ACM Trans. Graph.*, vol. 21, no. 3, pp. 693–702, Jul. 2002. [Online]. Available: http://doi.acm.org/10.1145/566654.566639

[8] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira, “Vr juggler: a virtual platform for virtual reality application development,” in *Virtual Reality, 2001. Proceedings. IEEE*, March, pp. 89–96.

[9] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, “A sorting classification of parallel rendering,” *Computer Graphics and Applications, IEEE*, vol. 14, no. 4, pp. 23–32, 1994.

[10] M. Makhinya, S. Eilemann, and R. Pajarola, “Fast compositing for cluster-parallel rendering,” in *Proceedings of the 10th Eurographics conference on Parallel Graphics and Visualization*, ser. EG PGV’10. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2010, pp. 111–120. [Online]. Available: http://dx.doi.org/10.2312/EGPGV/EGPGV10/111-120

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.