TerraME Observer: An extensible real-time visualization pipeline for dynamic spatial models

Antônio José C. Rorigues[1], Tiago G. S. Carneiro[1], Pedro R. Andrade[2]

[1] TerraLAB – Earth System Modeling and Simulation Laboratory,
Computer Science Department, Federal University of Ouro Preto (UFOP)
Campus Universitário Morro do Cruzeiro – 35400-000
Ouro Preto – MG– Brazil

[2] Earth System Science Center (CCST), National Institute for Space Research (INPE)
Avenida dos Astronautas, 1758, Jardim da Granja – 12227-010
São José dos Campos – SP– Brazil

aj.rodrigues@ymail.com, tiago@iceb.ufop.br, pedro.andrade@inpe.br

Abstract. *This paper presents ongoing research results of an extensible visualization pipeline for real-time exploratory analysis of spatially explicit simulations. We identify the software requirements and discuss the main conceptual and design issues. We propose a protocol for data serialization, a high performance monitoring mechanism, and graphical interfaces for visualization. Experiments for performance analysis have shown that combining multithreading and the BlackBoard design pattern reduces the visualization response time in 50%, with no significant increase in memory consumption. The components presented in this paper have been integrated in the TerraME modeling platform for simulation of terrestrial systems.*

1. Introduction

Computer modeling of environmental and social processes has been used to carry on controlled experiments to simulate the effects of human actions on the environment and their feedbacks (Schreinemachers and Berger, 2011). In these studies, simulated scenarios analyze issues related to the prognosis of amount and location of changes, which may support decision-making or public policies. Computer models are in general dynamic and spatially explicit (Sprugel et al., 2009; Wu and David, 2002), using remote sensing data and digital maps as inputs.

Dynamic spatially explicit models to study nature-society interactions, hereinafter referred as environmental models, are capable of generating a huge amount of spatiotemporal data in each simulation step. In addition, before any experiment, models need to be verified in order to fix logic faults. The sooner such errors are found, the sooner the implementation can be completed. Model verification and interpretation of simulation results can be more efficiently performed with the support of methods and tools capable of synthesizing and analyzing simulation outcomes.

Visualization components of environmental modeling platforms differ in the way they gather, serialize, and transmit state variable values to graphical interfaces. Such platforms may provide high-level languages to implement models or may be

delivered as libraries for model development in general purpose programming languages. In the latter situation, as in Swarm and RePast platforms, state variable values are available within the same runtime environment of graphical interfaces (Minar et al., 1996; North et al., 2006), making data gathering easier and faster. In the platforms that provide embedded languages, as NetLogo and TerraME, state variables are stored in this language memory space and need to be copied to the memory space where the graphical interfaces are defined (Tisue and Wilensky, 2004; Carneiro, 2006), i.e., to the memory space of a simulation core responsible for model interpretation and execution. This way, once collected, data needs to be serialized and transmitted according to a protocol that can be decoded by the graphical interfaces. As environmental modelers use to be specialists in the application domains (biologists, ecologists, etc) and do not have strong programming skills, this work focuses on modeling platforms that follow the second architecture.

As environmental simulations may deal with huge amounts of data, there might also be a huge amount of data that need to be transferred, which in turn can make the tasks of gathering, serializing, and transmitting data very time consuming. Land use change modeling studies discretize space in thousands or millions of regular cells in different resolutions, whose patterns of change need to be identified, analyzed and understood (Moreira et al., 2009). In these cases, the simulation could run on dedicated high-performance hardware, with its results being displayed on remote graphical workstations. Therefore, it might be necessary to transfer data from one process in this pipeline to the next through a network.

The main hypothesis of this work is that combining software design patterns and multithreading is a good strategy to improve visualization response times of environmental models, keeping the platform simple, extensible, and modular. This work presents the architecture of a high performance pipeline for the visualization of environmental models. It includes high-level language primitives for visualization definition and updating, a serialization protocol, a monitoring mechanism for data gathering and transmission, and several graphical interfaces for data visualization. This architecture has been implemented and integrated within the TerraME modeling and simulation platform (Carneiro, 2006).

The remainder of the paper is organized as follows. TerraME modeling environment is discussed in Section 2. Related works are presented in Section 3. Section 4 describes the architecture and implementation of the system, while experiments results are presented in Section 5. Finally, in Section 6, we present the final remarks and future work.

2. TerraME modeling and simulation platform

TerraME is a software platform for the development of multiscale environmental models, built jointly by the Federal University of Ouro Preto (UFOP) and the National Institute for Space Research (INPE) (Carneiro, 2006). It uses multiple modeling paradigms, among them the theory of agents, the discrete-event simulation theory, the general systems theory, and the theory of cellular automata (Wooldridge and Jennings, 1995; Zeigler et al., 2005; von Bertalanffy, 1968; von Neumann, 1966). Users can describe TerraME models directly in C++ or in Lua programming language (Ierusalimschy et al., 1996). TerraME provides several types of objects to describe

temporal, behavioral, and spatial features of models. *Cell*, *CellularSpace*, and *Neighborhood* types are useful to describe the geographical space. *Agent*, *Automaton* and *Trajectories* types represent actors and processes that change space properties. *Timer* and *Event* types control the simulation dynamics. During a simulation, the Lua interpreter embedded within TerraME activates the simulation services from the C++ framework whenever an operation is performed over TerraME objects. The TerraLib library is used for reading and writing geospatial data to relational database management systems (Câmara et al., 2000). The traditional way to visualize the outcomes of a simulation in TerraME is by using the geographical information system TerraView[1]. However, TerraView cannot monitor the progress of simulations in real-time.

3. Related Works

This section compares the most popular simulation platforms according to services related to graphical interfaces to visualize simulation outcomes, including the extensibility of such interfaces. Major environmental modeling platforms provide graphical interfaces for visualization. However, their visualization components work as black boxes and their architectural designs have not been published. Swarm and Repast are multi-agent modeling platforms delivered as libraries for general purpose programming languages (Minar et al., 1996; North et al., 2006). They provide specific objects for monitoring and visualization. New graphical interfaces can be developed by inheritance. Their monitoring mechanism periodically updates interfaces in an asynchronous way, i.e., simulation runs in parallel with visualization interfaces; it does not stop waiting for interface updating.

NetLogo is a framework that provides tools for multi-agent modeling and simulation (Tisue and Wilensky, 2004). Models are described in a visual environment focused in building graphical user interfaces by reusing widget components in a drag-and-drop fashion. Rules are defined in a high-level programming language. Model structure and rules are translated into a source code in a general purpose programming language, which is finally compiled. Communication between simulation and graphical interfaces is also asynchronous. Graphical interfaces can be periodically updated or explicitly notified by the implementation.

4. Architecture and Implementation

This section describes computer systems and methods employed to achieve our goals. We identify the main requirements of an environmental model visualization pipeline, discuss the design of visualization pipeline and graphical interfaces, present the high-level language primitives used to create visualizations and to associate them to model state variables, formally define the serialization protocol, and detail the object oriented structure of the monitoring mechanism.

4.1. Software requirements

Some requirements have been considered essential to a visualization pipeline for real-time exploratory analysis of spatially explicit dynamic models.

1 http://www.dpi.inpe.br/terraview/

- Functional requirements: graphically present the dynamics of continuous, discrete and spatial state variables; provide visualizations to temporal, spatial and behavioral dimensions of an environmental model; graphically exhibit the co-evolution of continuous, discrete and spatial state variables so that patterns can be identified and understood.
- Non-functional requirements: present real-time changes in state variables with as little as possible impact on the simulation performance; enable the monitoring mechanism to be extensible so that new visualizations can be easily developed by the user; keep compatibility with models previously written without visualizations.

4.2. Monitoring mechanism outline

The visualization pipeline designed consists of three main stages: recovery, decoder, and rendering. Recovery stage gathers the internal state of a subject in the high-level language and serializes it through the protocol described in section 4.3. Decoder stage deserializes the data. Finally, rendering stage generates the result image, as shown in Figure 1.

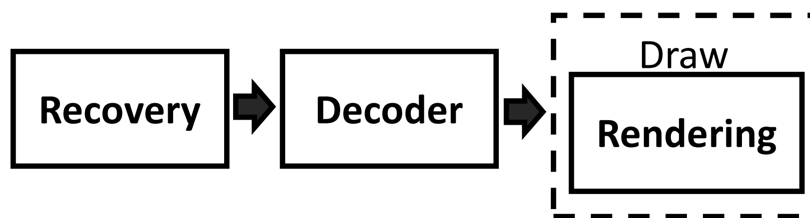

Figure 1. Visualization pipeline (Adapted from [Wood et al 2005])

The monitoring mechanism is structured according to the Observer software design pattern (Gamma et al., 1995). Graphical interfaces for scientific visualization are called *observers* and present real-time changes in the internal state of any TerraME object. Each instance of a model component within an observer is called *subject*. As Figure 2 illustrates, several observers can be linked to a single subject, so that its evolving state can be analyzed simultaneously in many ways. Changes in a subject need to be explicitly notified to the observers in the source code. This assures that only consistent states will be rendered by the observers and gives complete control to the modeler to decide in which changes he is interested. When notified, each observer updates itself requesting information about the internal state of its subject. Then, the state is serialized and transferred to the observers to render the graphical interface.

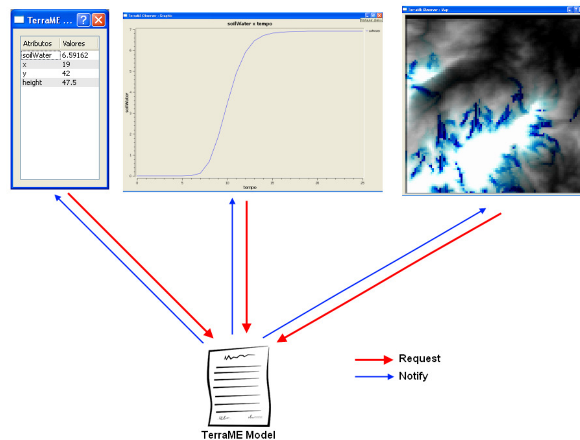

Figure 2. Monitoring mechanism is structured according to the Observer software design pattern

Graphical interfaces and state variables might potentially exist in the memory space of different processes. In TerraME, state variables are stored in Lua during the simulation, with observers being defined in the C++ simulation core, as illustrated in Figure 3. Each observer is implemented as a light process (thread) avoiding interfaces to get frozen due to some heavy CPU load. The *blackboard* software design pattern has been integrated within the monitoring mechanism to intermediate communication between subjects and observers (Buschmann, 1996). Blackboard acts as a cache memory shared by observers in which the state recovered from the subjects are temporarily stored to be reused by different observers. This way, it is maintained in the same processes of the observers. This strategy aims to reduce the processing time involved in gathering and serializing state variable values, as well as the communication between subjects and observers.

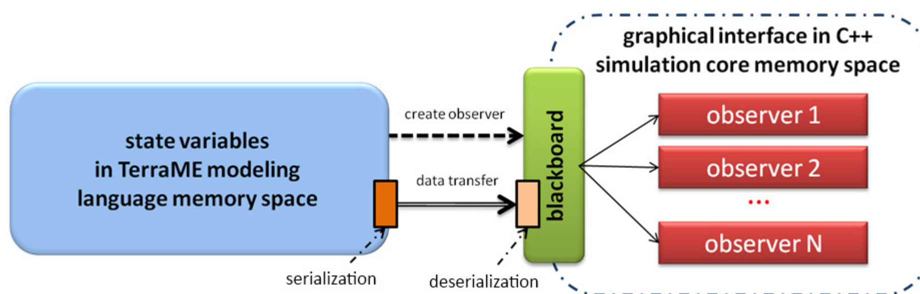

Figure 3. Monitoring mechanism general architecture

4.3. Serialization protocol

Observers are loosely coupled to the subjects. The communication between them is performed through the serialization protocol whose message format is described using the Backus-Naur formalism as follows.

```
<subject> ::= <subject identifier> <subject type> <number of attributes>
         <number of internal subjects> [*<attribute>] [*<subject>]

<attribute> ::= <attribute name> <attribute type> <attribute value>
```

A subject has a unique ID, characterized by its type and an optional sequence of attribute. It is recursively defined as a container for several optional internal subjects. The protocol allows the serialization of a complete subject or only the changed parts, saving communication and processing time. Extending TerraME with new observers requires only decoding these messages and rendering their content, no matter how subjects have been implemented.

4.4. Monitoring mechanism detailed structure

Figure 4 shows the class diagram of the monitoring mechanism and Figure 5 shows how the interactions between objects of these classes take place. A dirty-bit has been added to each element in the blackboard and to each observer. It indicates whether the internal state of the associated subject has changed, pointing out that such objects need to be updated to reflect the new state. Thus, when the modeler notifies the observers about changes in a subject, this notification only sets the dirty-bits to true. When an observer requests data about a dirty subject stored in the blackboard, the latter first updates itself, sets its dirty-bit to false, and then forwards the data to the observer. All others observers that need to be updated will find the data already decoded, updated, and stored in the blackboard. This way, a subject is serialized only once, even when there are many observers linked to it. After rendering the new subject state, an observer sets it dirty-bit to false to indicate that the visualization is updated.

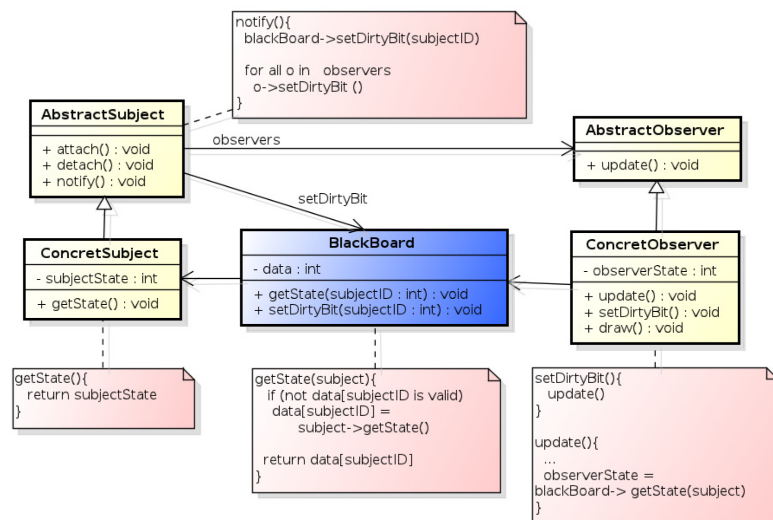

Figure 4. Class diagram of monitoring mechanism - integration between Blackboard and Observer design patterns

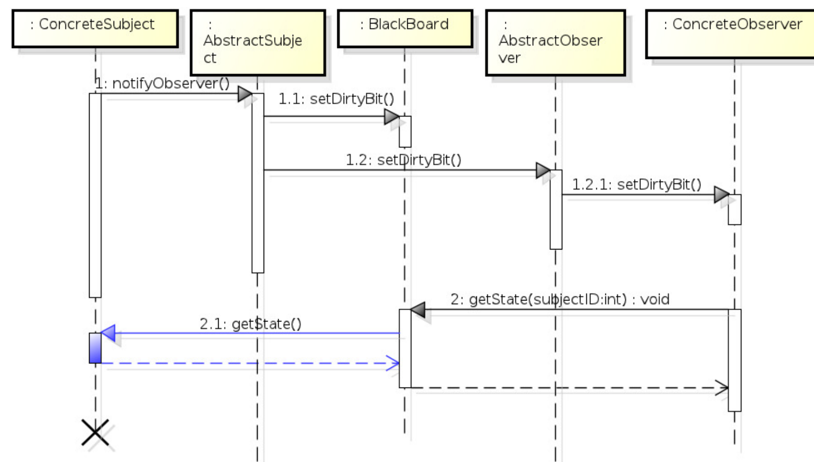

Figure 5. Sequence diagram of monitoring mechanism– interaction between Observer pattern and BlackBoard design patterns

4.5. TerraME observers

Several types of observers have been developed to depict the dynamics and the co-evolution of discrete, continuous, and spatial state variables. The left side of Figure 6 illustrates a dynamic table and a dynamic dispersion chart showing attributes of a single Cell. An attribute is an internal variable or property of some object, such as the size of a CellularSpace object and the state of an Agent. The right side shows two different time instants of an observer map that displays a CellularSpace. The amount of water in the soil is drawn from light blue to dark blue over the terrain elevation map drawn from light gray to dark gray. This way, the modeler can intuitively correlate the dynamics of the water going downhill with the terrain topography.

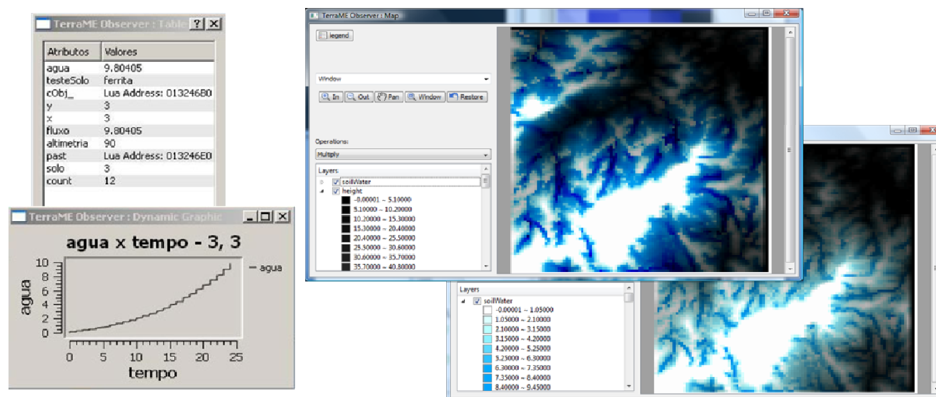

Figure 6. Different types of TerraME observers: dynamic tables, charts and maps

4.6. Monitoring mechanism programming interface

In order to create an observer and attach it to a subject, the modeler must explicitly declare an *Observer* object. The following command creates the “myObs” observer to monitor the attribute called *soilWater* from the subject “myCell”:

```
myObs = Observer{
    type = "chart",
    subject = myCell,
    attributes = {"soilWater"}
}
```

The parameter *type* is a string indicating which observer will be used, while the parameter *subject* is a TerraME object. Each type of subject can be visualized by a predefined set of observer types. The architecture is also flexible enough to allow the modeler to create new observer types, extending the C++ abstract class named *AbstractObserver*. The parameter *attributes* is a table of subject attributes that will be observed. Once created, the observer is ready to show the states of its subject. Each time the modeler wants to visualize the changes in a subject, rendering all observers linked to it, he must explicitly call the function *notify()* of this subject.

5. Performance analysis

Experiments were conducted to evaluate the performance of the visualization pipeline. These experiments measure the memory consumption and the response time involved in visualization interface updating. They also identify system bottlenecks, depicting the service time of each stage of visualization pipeline. The response time includes:

(1) Recovery time, which is spent to gather state variables values in the high-level language memory space, serializes according to the protocol message format (section 3.6) and transfers serialized data to the blackboard;

(2) Decode time, which is consumed to deserialize the message;

(3) Waiting time, which is the time elapsed between the instant that a subject request observers update by calling its notification function and the instant that this request starts to be served by the first observer thread to arrive in the CPU; and

(4) Rendering time, which the period of time consumed to map data in a visual representation and display it in graphical interfaces.

As described in Table 1, four experiments were performed, varying the type of subject, the number of monitored attributes and the number and type of observers. The experiments use an atomic type (Cell) and a composed type (CellularSpace). In experiments 1 and 2, a subject Cell with 2 attributes and 12 attributes, respectively, was visualized by several chart observers. In experiments 3 and 4, a CellularSpace with 10000 cells was visualized by 2 map observers and several map observers, respectively. This workload evaluates the impact of using blackboard to recover data, reducing the communication channel by reusing the decoded data.

Experiments were performed in a single machine, a 64 bits Xeon with 32 GBytes of RAM using Windows 7. Each experiment was repeated 10 times and averaged by memory consumption and the amount of serialized bytes. In each experiment, 100 simulation steps were executed and observers were updated at the end of each step.

Table 1 – Workload of the performance analysis experiments

Experiment	Subject	Attributes	Observer
1	Cell	2	2 charts
2	Cell	12	12 chart
3	100 x 100 CellularSpace	3	2 maps
4	100 x 100 CellularSpace	13	12 maps

Figure 7 presents the results comparing the simulations with and without blackboard (BB) as cache memory. It shows that the blackboard reduces significantly the number of serialized bytes, because attributes are serialized in the first data request and subsequent observers retrieve this data directly from the cached blackboard.

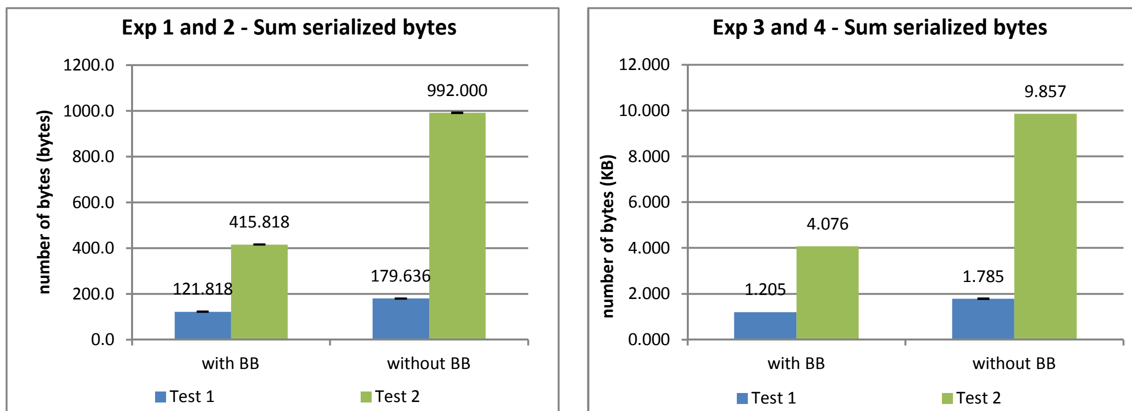

Figure 7. Amount of raw data serialized per notification in each experiment.

Figure 8 shows the average response time of experiments 1 and 2 decomposed in the times of each stage of the visualization pipeline. We can see that the rendering is most time consuming component. Comparing results of experiments 1 and 2 is possible to infer that the number of attributes being observed has a considerable impact on the average response time. However, there is no advantage in using blackboard with very small subjects.

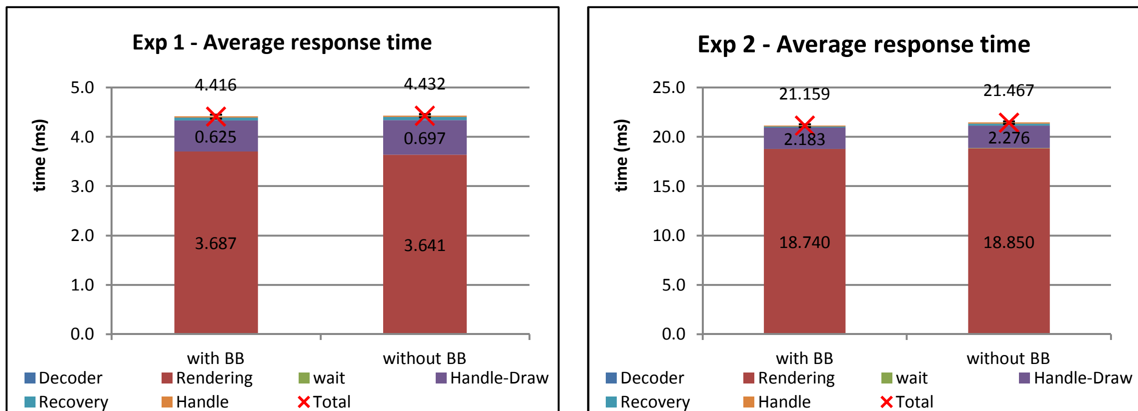

Figure 8. Average response time of experiments 1 and 2.

Figure 9 shows the average response time of experiments 3 and 4 decomposed in the service times of each stage of the visualization pipeline. Note that blackboard can significantly decrease the average response time in the visualization of large objects.

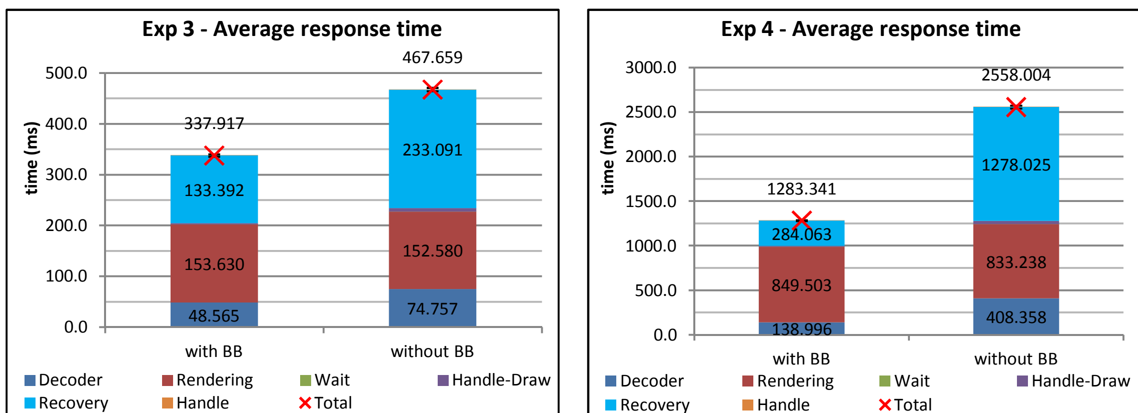

Figure 9. Average response time of experiments 3 and 4.

Figure 10 shows the average memory consumption of each experiment. It is possible to see that using blackboard does not bring any significant increase in memory consumption.

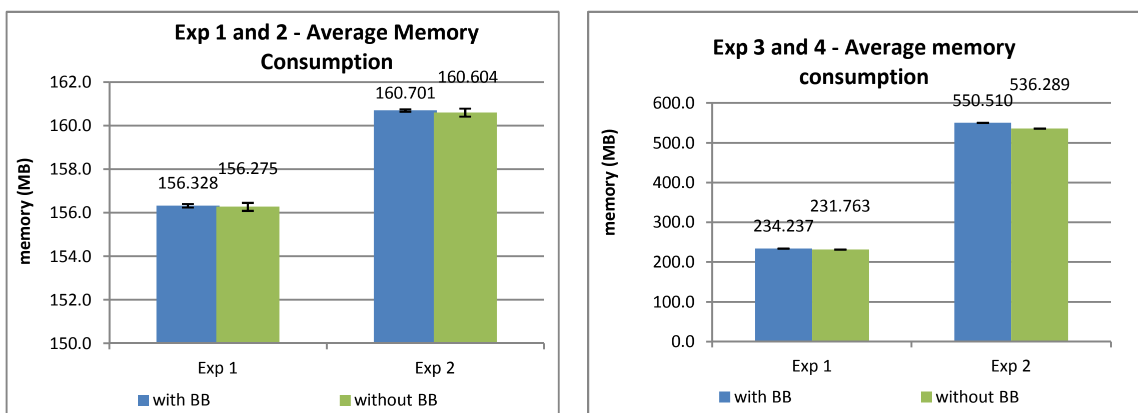

Figure 10. Average memory consumption of each experiment.

6. Final Remarks

In this work, we describe an extensible visualization component for real-time monitoring of environmental simulations. We demonstrate that combining multithreading and blackboard is a good technique to improve visualization performance, significantly decreasing the visualization response time with no expressive increase in memory consumption. The developed graphical interfaces are able to render discrete, continuous and spatial state variables of environmental models written in TerraME, rendering instances of all TerraME types. Visualizations are also able to graphically exhibit the co-evolution state variables, allowing the understanding of how a variable influences other and help identify some logic faults. The monitoring mechanism can be easily extended by inheritance. New observer types can also be created using the same mechanism. The new visualization capabilities added to TerraME do not affect models previously written in this modeling platform, keeping backward compatibility. Consequently, the proposed visualization mechanism satisfies all functional requirements stated in section 4.1.

Future works include adding a synthesis stage to the visualization pipeline. In this new stage, it will be possible to apply filters and statistical operations to raw data to make data analysis easier. It is also necessary to implement change control algorithms. New experiments will be performed to measure performance by transmitting only objects and attributes that have changed along the simulation. Other experiments will evaluate the impact of the blackboard and of compression algorithms in a client-server version of the proposed visualization mechanism. Initial evaluation of the client-server version has shown that the use of blackboard on the client side reduces the exchange of messages by half using TCP protocol. Finally, experiments will be conducted to quantitatively compare the visualization mechanisms of the most relevant modeling platforms with the one presented in this work.

Acknowledgements

The authors would like to thank the Pos-Graduate Program in Computer Science and the TerraLAB modeling and simulation laboratory of the Federal University of Ouro Preto (UFOP), in Brazil. This work was supported by the CNPq/MCT grant 560130/2010-4, CT-INFO 09/2010.

References

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc.

Câmara, G., Souza, R., Pedrosa, B., Vinhas, L., Monteiro, A.M., Paiva, J., Carvalho, M.T., Gattass, M., (2000). TerraLib: Technology in Support of GIS Innovation, II Brazilian Symposium on Geoinformatics, GeoInfo2000: São Paulo.

Carneiro, T. G. S. (2006). *Nested-CA: a foundation for multiscale modeling of land use and land change.*. Ph.D Thesis, INPE - Instituto Nacional de Pesquisas Espaciais, Brazil, Computação Aplicada.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional.

Ierusalimschy, R., Figueiredo, L.H., Celes, W., (1996). Lua-an extensible extension language. Software: Practice & Experience 26(6) 635-652.

Minar, N., Burkhart, R., Langton, C., Askenazi, M., (1996). The Swarm Simulation System: A Toolkit for Building Multi-Agent Simulation. SFI Working Paper 96-06-042

Moreira, E.; Costa, S.; Aguiar, A. P.; Câmara, G., Carneiro, T. G. S., (2009). Dynamical coupling of multiscale land change models Landscape Ecology, Springer Netherlands, 24, 1183-1194

North, M.J., Collier, N.T., Vos, J.R., (2006). Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit. ACM Transactions on Modeling and Computer Simulation 16(1) 1-25.

Schreinemachers, P. and Berger, T. (2011). An agent-based simulation model of human-environment interactions in agricultural systems. *Environmental Modelling & Software*, 26(7):845 – 859.

Sprugel, D. G., Rascher, K. G., Gersonde, R., Dovciak, M., Lutz, J. A., and Halpern, C. B. (2009). Spatially explicit modeling of overstory manipulations in young forests: Effects on stand structure and light. *Ecological Modelling*, 220(24):3565 – 3575.

Tisue, S., Wilensky, U., (2004). NetLogo: A Simple Environment for Modeling Complexity, International Conference on Complex Systems: Boston.

von Neumann, J., (1966). Theory of Self-Reproducing Automata. Edited and completed by A.W. Burks., Illinois

Wood, J.; Kirschenbauer, S; Döner, J.; Lopes, Adriano and Bodum, L. (2005). Using 3D in Visualization. In: Dykes, J; Maceachren, A. M.; Kraak, J. (Eds.). *Exploring Geovisualization*. Elsevier. p. 295-312.

Wooldridge, M.J., Jennings, N.R., (1995). Intelligent agents: Theory and practice. Knowledge Engineering Review 10(2).

Wu, J. and David, J. L. (2002). A spatially explicit hierarchical approach to modeling complex ecological systems: theory and applications. *Ecological Modelling*, 153(1-2):7 – 26.

Zeigler, B.P., Kim, T.G., Praehofer, H., (2005). Theory of modeling and simulation. Academic Press, Inc., Orlando, FL, USA.