3D objects visualization for remote interactive medical applications

Johan Montagnat, Eduardo Davila, Isabelle Magnin

To cite this version:

Johan Montagnat, Eduardo Davila, Isabelle Magnin. 3D objects visualization for remote interactive medical applications. 2010. hal-00476888

HAL Id: hal-00476888
https://hal.science/hal-00476888

Preprint submitted on 27 Apr 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

3D objects visualization for remote interactive medical applications

Johan Montagnat, Eduardo Davila, Isabelle E. Magnin
CREATIS, CNRS UMR 5515
INSA Lyon, bâtiment Blaise Pascal
20 avenue Albert Einstein
69621 Villeurbanne Cedex, France
http://www.creatis.insa-lyon.fr/

Abstract

3D medical images produced in health centers represent tremendous amounts of data spread over different places. Automatic processing of these data often allow quantitative and repeatable measurements on large databases that are valuable for diagnosis and pathology studies. Grid technologies offer large and distributed computing power suited to medical images analysis. However, the need for human control prevent complete off-line execution of some algorithms. Interactive programs with user supervision are made difficult on a grid architecture due to the remote program execution. In this paper we propose a new framework for interactive execution of remote applications dealing with large 3D medical data sets. Our solution is both efficient and easy to use from the user point of view. It can adapt to almost any medical application requirements. The formal architecture is presented and a prototype is demonstrated.

1. Context

For years now, health centers have been using an increasing number of digital 3D sensor for medical data acquisition [1], such as Computed Tomography scanners (CT), Magnetic Resonance Imagers (MRI), Positon Emission Tomography scanners (PET), Single Photon Emission Computed Tomography (SPECT), and recently 3D ultrasound devices (US). The 3D images, or time sequences of 3D images, produced represent tremendous amount of data for which manual inspection and/or processing is tedious, error prone, and subject to inter-operator variability.

In the past decade, an intensive research effort aiming at automatically analyzing 3D medical images was developed. Dedicated 3D image processing algorithms allow quantitative analysis of medical data repeatedly. They also allow mass data processing for large databases analysis as needed by epidemiological studies for instance. However, many of these algorithms still require human supervision for assessing the processing validity and for responsibility reasons. Therefore, many algorithms propose an interface for interactive visualization and guidance by a human operator [6].

Independently, grid technologies have been developing for the past few years [8]. The idea is to assemble a wide network of computers that can be seen from the user point of view as a single computational unit. A middleware layer provides facilities to submit jobs [4], store data [5], and get information from the grid while hiding the underlying grid architecture to the user and physical machines used for storage and computations. Grid technologies offer a platform with many added value for medical image processing, including:

- an increasing computing power for complex algorithms, such as geometrical and physiological modeling of human organs, that require large amounts of memory and processing resources;
- a distributed platform with shared resources for different medical partners with similar needs in their data processing;
- a common architecture to access heterogeneous data and algorithms for processing;
- and the ability to process very large amounts of data.

However, grid processing introduce some constraints on applications as they are executed on remote machines.

The work presented in this paper is developed as a contribution to the European DataGrid project (EDG)[1] although it is not limited to this architecture and it intends to provide a generic solution for interaction with remote applications. The EDG grid is made of computing units spread in different sites all over Europe (and even beyond) and a middleware layer proposing grid services. Our problem is to

[1] http://www.eu-datagrid.org

port interactive applications on this distributed architecture. Interactive applications have a user interface that should appear on the user interface machine while the application itself is running on a remote computing element. However, direct usage of an X client/server si not admissible for two reasons:

- X transmission of graphic bitmaps is network intensive (as compared to network transmission of minimal geometric information) and it is not adapted to a high degree of interactivity;
- 3D graphic visualization is needed and it is usually achieved locally by the user workstation graphic hardware (i.e. contrarily to [7] the remote machine is not supposed to have specific graphic capabilities).

Specific developments of an interactive application architecture are therefore needed. The communication between the interface and the computation program should remain as transparent as possible from the programmer point of view to ease the integration of grid-aware medical image processing applications.

Section 2 details medical applications and the issues arising when porting such applications to the grid. Section 3 describes the proposed solution. Finally, section 4 gives prior results on a sample application. In the following paper, the *local* machine corresponds to the user workstation on which the graphic interface should appear and the *remote* machine corresponds to the grid computation node.

2. Grid-aware medical applications

2.1. 3D data in medical applications

Medical image processing applications manipulate 3D images, or time sequences of 3D images, and 3D models. Most interactive application require the display of one (or more) 3D image on which is overlaid some information proper to the application (e.g. a geometrical model of an anatomical structure). For instance, figure 1 illustrates three different applications. On the left is an MRI slice showing the aorta on which is overlaid a vessel model used for the aorta segmentation. On the right is a 3D mechanical model of the heart left and right ventricles used for heart image registration (courtesy of Q.-C. Pham and P. Clarysse, CREATIS). At the bottom is diagrammed a surgery simulator composed by an interactive console (haptic device and graphic rendering) communicating with a computation server on which is stored an organ bio-physical model.

Visualization of 3D medical images in not straightforward since a user cannot at a glance visualize the interior of a 3D volume. Different visualization techniques exist that transform the original volume in a representable data. This

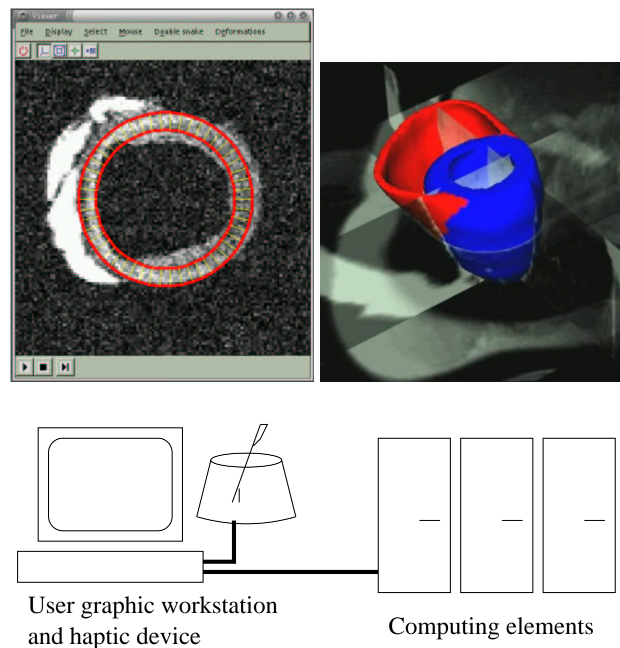

Figure 1. Three interactive medical application examples (see text).

might be a single slice extracted out of the volume as illustrated in top row of figure 1 or a synthesized representation as may be obtained by surface or volume rendering techniques for example [3].

Application dependent additional information usually involve geometrical representation of some internal modeling of the data. This information is often much more compact than the complete 3D data set. Therefore, rather few information need to be exchanged between the computing machine and the graphic interface.

In the literature, many works dealing with remote data visualization are reported. Many work concern post-processed data visualization [10], while other consider interaction with the data [3]. Some of them offer asynchronous objects management and real-time constraints [12, 9]. However, few of them address the specific needs of medical image processing applications [6], and optimized computing grids for solving medical problems is still an emerging field.

2.2. Interactive applications

Interactive applications as depicted above are based on a specific structure. The body of the program in made of an infinite loop as illustrated in figure 2.

1. The user sends some input (interactivity result) to the

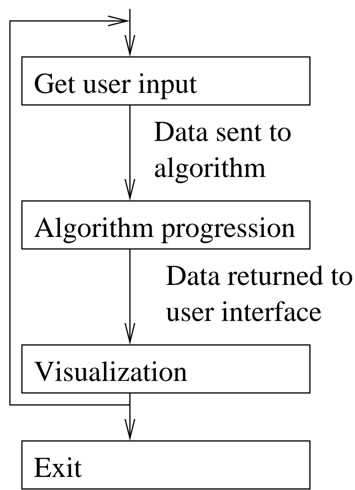

Figure 2. Interactive application general structure.

program. This step is non-blocking (no user input may be needed at some point) and might be asynchronous in some cases.

2. The algorithm progresses taking into account the user input.

3. The updated status of the computation is displayed in the user interface to allow further evaluation/interaction with the algorithm.

Some data need to be exchanged between the algorithm and the user interface on a periodic basis in this loop: when transmitting user inputs to the algorithm and when updating the user interface to reflect the changes in the algorithm status.

2.3. Issues in porting interactive applications on grid architectures

On a grid architecture, the user is submitting jobs to the grid using the grid middleware interface. Its jobs are executed on any suitable machine composing the grid without any prior information on the selected target. The application should therefore be able to connect to the user interface in order to exchange user input and graphic information using a communication protocol. In the following section, we propose a generic architecture for interactive applications.

To obtain a high interactivity level, e.g. including realtime requirements such as needed for the surgery simulation application depicted above, communications between the program and the user interface should be as efficient as possible. For some critical applications, a guaranteed network quality of service may even be mandatory.

The communication protocol should be designed to make the user's code as independent as possible of the remote interface system. This is mandatory to ease the transcription of existing applications and to encourage new developments in the grid framework.

3. Remote 3D visualization and interaction framework

Our implementation is based on C++ libraries developed in our laboratory, although the proposed framework does not depend on this specific language. We first describe the local application framework as it exists on stand-alone machines and we then present its adaptation to the remote application framework.

3.1. Stand-alone application

A set of base classes is provided to ease construction of interactive medical applications. Concerning user interface, basic 2D and 3D displaying functionalities are provided as illustrated in figure 3.

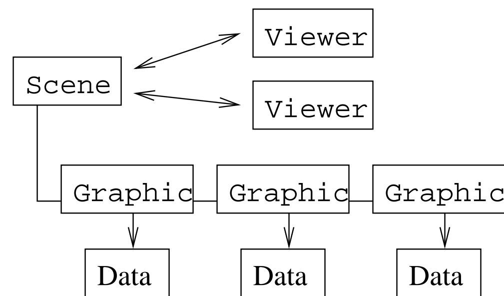

Figure 3. Graphic functionalities provided to the user.

A `Graphic` base class is provided. It is responsible for interaction between the graphic system and the user objects. In particular, a `redraw` virtual method is provided to be redefined by each graphics component. This method is called each time the user interface need to be refreshed.

A `Scene` is composed of a list of graphic objects. A scene defines a common frame to assemble different graphic objects in space. Two different scenes represent to different worlds without interactions.

Finally, a `Viewer` represents a window that displays the graphic content of a scene on the screen. Several viewers may visualize a same scene with different viewing parameters. The viewers allow interaction with user objects by

different means (contextual menus, mouse events forwarding...).

The graphic interface is completely disconnected from the user objects. Therefore an application does not depend on a graphic system and may be recompiled to execute off-line, without interface. To achieve this, the user writes the object composing its application data separately from the graphic object (that inherits from `Graphic`). To each data object is associated a graphic object, but the data object has no knowledge about its graphic interface.

The application itself is made of a main loop as illustrated in figure 2. The user can interact with its data through the graphic interface. Periodically, the user input are sent to the algorithm. The algorithm progresses and then ask for an update of the scene. This causes all graphics objects to be redrawn in all windows.

3.2. Remote Visualization

In order to achieve remote visualization, the data and the graphic objects should lie on two different machines communicating together: a (local) interface program contains graphic information while a (remote) computation program contains the data. The overall architecture is diagrammed in figure 4.

On the user interface machine, a graphic daemon is waiting for connections on a given port. The user first submits an interactive job from a shell using the grid middleware interface (1). Once executing, the remote process connects back to the graphic daemon of the user interface (2). The graphic daemon then forks a new interface process (3) to handle interactions with the computational process. A communication channel is opened between the two processes (it may be a simple socket or something more sophisticated such as a CORBA bus [11] or CCA ports [2] to handle platforms heterogeneities).

Once the two processes have started and the communication channel is opened, the local program creates a graphic interface and enters a main loop where it iteratively processes graphic events and communication events until it receives an exit message from the computation program. The computation program sends a description of the desired interface and then enters the interactive loop that was diagrammed in figure 2.

Note that in this framework, the interface program is generic and does not depend on a precise application. The graphic interface description is sent by the computational program to the user interface. This is needed because the program may be available from the grid, without the user having to pre-install anything on its machine. The only mandatory component on the user interface is the graphic daemon on which the remote application connects. Technically, the interface may be described by different means such as using java byte code sent through the communication channel. In our implementation we rely on a platform-independent windowing system (wxWindows[2] wrapped in the python[3] scripting language). The script is sent through the channel and executed on the local machine.

This solution also proves to be very flexible. A program can modify its graphical interface dynamically depending on the algorithm evolution and the user interactions. There is not limitation on the generated interface induced by the system. The internal mechanism (see below) for creating the interface is completely hiding the communication layer to the user. A transparent protocol is defined to communicate graphic object rendering commands to the user interface. It eases the development of user applications and allow to run the program stand-alone or remotely without any change in the user code.

3.3. Communication protocols

When comparing the general remote visualization scheme of figure 4 with the general local application scheme of figure 3, it appears that objects should be created on different machines. Indeed, the communication channels comes between the graphic and the data objects. All viewers, scenes, and graphic objects code is executed on the local host while the data objects are part of the remote process. Figure 5 details a remote application structure.

The user program initializes an interface using a scripted description and a display parameter. The system initialization method provided connects to the given machine and the remote graphic daemon forks an interface process. Communicating objects are automatically created on both sides and communication channels are opened. For the sake of clarity, two different communication channels are opened with two different controllers. The first one is used for interface control (interface commands and interaction feedback). The second one is responsible for mapping data from remote data objects to graphic representations on the local machine.

3.3.1 Interface protocol

The user program first sends the interface script and can then create data and graphic objects using the standard C++ `new` operator. In a stand-alone application (figure 3), the user interface script is evaluated locally and the graphic objects are created on the local machine. In a remote application, the interface is sent to the local machine and the graphic object constructor causes a remote graphic object creation. The remote graphic object is identical to the local graphic object so that the user does not need to change

[2] `http://www.wxwindows.org`
[3] `http://www.python.org`

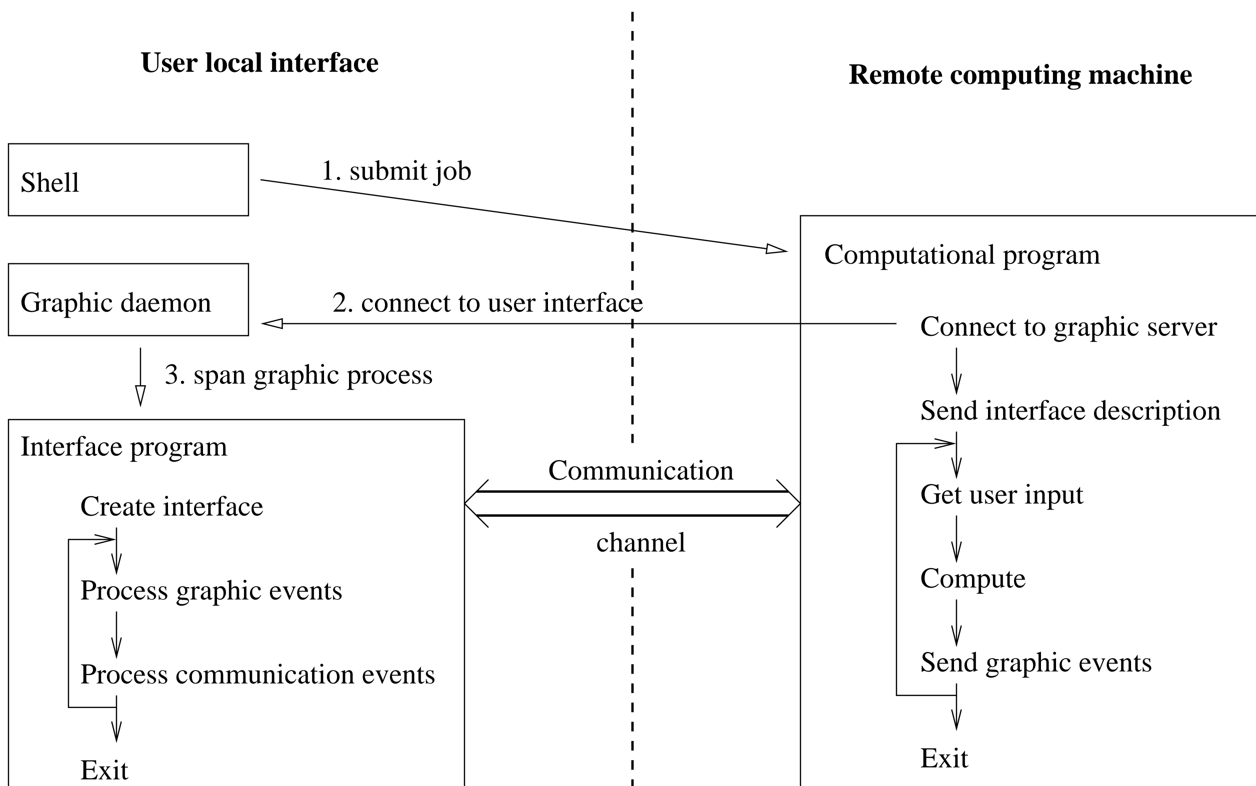

Figure 4. Remote interactive application general diagram (see text).

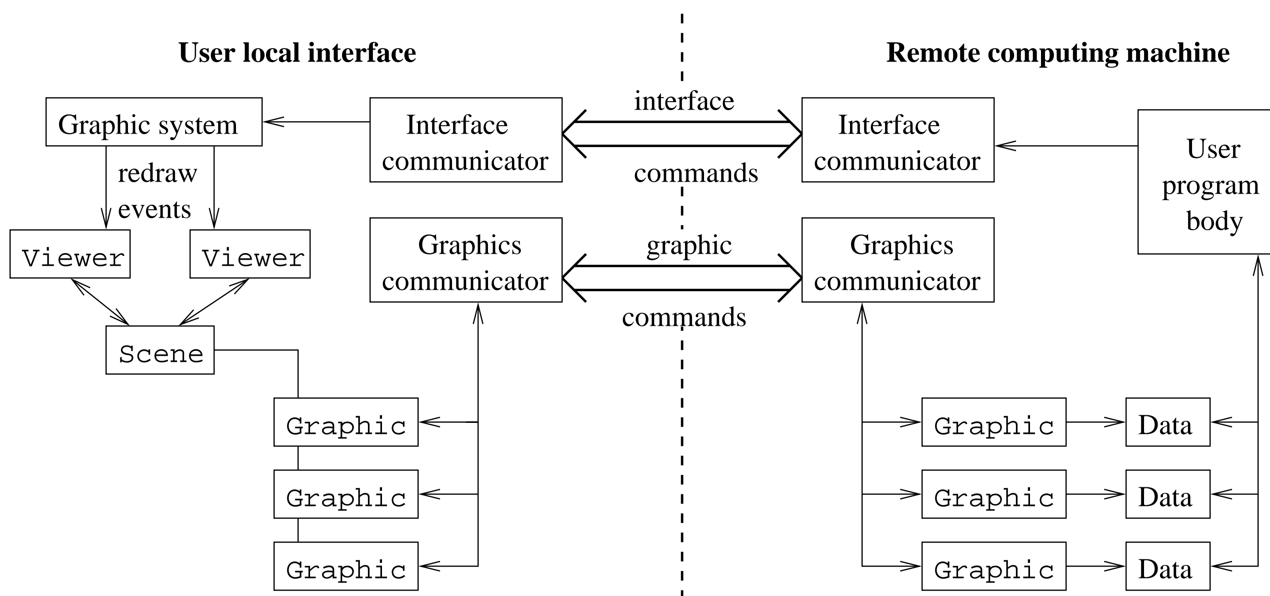

Figure 5. Remote interactive application detailed diagram (see text).

its code whether running stand-alone or remotely. To guarantee minimal prior installation on the user workstation, the graphic object code should be sent from the remote program to the interface dynamically upon object creation. This may be done using java bytecode or dynamic libraries for instance.

An interface protocol allows the user to create or delete interfaces and objects. The protocol itself is hidden behind standard method calls such as `createInterface(`*interface-script.py*`)`. The underlying system exchanges pre-defined messages between the computing and the local machine Messages are interpreted and proper action executed by the interface communicator objects. Messages sent from the remote to the local machine include:

- TERMINATE. Disconnect and terminate interface program.
- EVALUATE. Evaluate transmitted script.
- CREATE. Create a new graphic object.
- DELETE. Delete a graphic object.

Conversely, messages sent from the local to the computing machine include:

- TERMINATE. Disconnect and terminate computation program.
- READ. Read new data from a file/stream.
- WRITE. Write data to a file/stream.
- START. Start computation loop.
- STOP. Stop computation loop.

As illustrated in figure 4, the interface communicators on the local and the remote machines periodically poll the incoming messages to respond these commands.

3.3.2 Graphic protocol

Whenever a viewer window needs to be refreshed (because the viewing parameters changed or the graphic window needs to be redrawn), the graphic system if the local process sends redraw events that cause the `redraw` method of all graphic objects to be called. This method is responsible for displaying some data and therefore needs to retrieve the data content on the remote machine. A second communication protocol is needed. These communication are blocking: all graphic objects first retrieve their needed data, then the graphic interface is updated.

Since the graphic objects on the local and the remote machine are identical, the local graphic only needs to send one INIT message to initiate data transmission. When receiving this message, the remote graphics communicator causes the remote graphic object `redraw` method to be called. Each time the remote object fetches some data from the data object, it uses a method that sends the retrieved information to the local host. Each time the local object fetches the same data, the it waits for the information to arrive from the remote object.

This is clarified in the following simple example. Suppose that the data object is a polygonal contour. It contains an array of position vectors. The pseudo-code corresponding to the `redraw` method on the local and the remote host would look like:

```
void Polyon::localRedraw() {
  int n = getInteger();
  Vector v1 = getVector();
  for(int i = 1; i < n; i++) {
    Vector v2 = getVector();
    draw(v1, v2);
    v2 = v1;
  }
}

void Polyon::remoteRedraw() {
  int n = data->getNbVertices();
  sendInteger(n);
  for(int i = 0; i < n; i++)
    sendVector(data->getVertex(i));
}
```

The corresponding communication diagram is shown in figure 6. The INIT message precedes the entrance into the local redraw code and causes the remote redraw code to start. Only SEND messages from the remote to the local machine are then needed. Both functions end with implicit mutual consent. From the user point of view, message exchanges are hidden in the get/send function calls. Get function calls appear exactly in the same order than send function calls. Get and send functions are defined for every primitive types or array of primitive types needed for exchanging geometrical information (language primitive types, vectors, matrices...).

On a stand-alone execution, both “local” and “remote” code are executed on the same machine. The “remote” code is first executed. To avoid the overhead of a communication channel, the send methods are redefined to push the addresses of “sent” data on a stack. The “local” code then pop the data out of the stack during the get methods. For efficiency reasons, a send function that returns a pointer on a data set acts differently when it is executed stand-alone or remotely. On a remote system, the pointed data have to be transmitted through the communication channel and therefore copied in a temporary buffer on the remote machine.

On a stand-alone execution, only the pointer to the original data is pushed on the stack thus avoiding a useless copy.

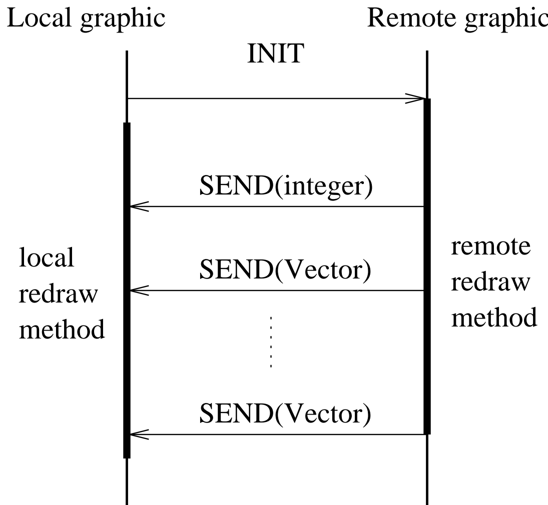

Figure 6. Communication messages between a local and a remote graphic object.

3.4. Interaction with remote applications

The user can interact with the data either by direct mouse action in the graphic window (to select a graphic object for instance) or through graphic controls (menus, buttons...). Our graphic interface proposes 4 different modes to interpret mouse actions (button press and pointer moves) occurring in a viewer window:

- Viewer mode. The viewing parameters are controlled by mouse motion.
- Selection mode. A graphic object is selected by a mouse click.
- Object motion. The selected object is moved according to mouse motion.
- Object mode. Mouse controls are captured by the selected object for its internal use.

The *viewer mode* is completely handled on the local machine as viewers code is locally available.

In *selection mode* a mouse click selects the "closest" object from the mouse pointer. Similarly to the `redraw` method a `distanceTo` method of each graphic may be redefined by the user to determine the distance between a mouse click and each graphic object. When a mouse click occurs in a viewer window, a SELECT message is sent through the communication channel to each remote graphic object causing their `distanceTo` method to be evaluated on the remote node (since this method usually involve accessing the data object). Distance values are returned and the object corresponding to the minimal distance is selected.

The *object motion* mode similarly involves code execution on the remote machine to update the data. On the local machine, a translation vector, a scale factor, or a rotation matrix is estimated depending on the mouse motion. TRANSLATE, SCALE, and ROTATE messages are sent to the remote nodes. Data is remotely updated and a REDRAW message is sent back to redisplay the moved object.

When selected, a graphic object also pops up some object-related menu entries and controls. Figure 7 shows object-related controls for a 3D image renderer. These menus and controls are created by the graphic object and are therefore transmitted to the local host with the graphic object. Most menu entries and controls act directly on the graphic object parameters and are handled locally. However, some of them may cause changes to happen on the represented data. Since these changes are data dependent, it is important that the graphic communicator interface let the user define its own messages. Thus the system is extensible and the user can add any additional message he needs in its application. A new message is composed of a unique identifier (given by the graphic system on new message creation), a description of parameters to transmit, and an associated method of the remote graphic object. The associated method has a single object parameter that contains a list of transmitted parameters. When the control (menu, button...) is activated, the data to transmit are sent with the message, causing the remote method to get called on the remote node. For transparency reasons, the system proposes a single method to bind a control action with a graphic method. This method creates the new message structure in case of remote execution or simply binds the associated method with the control for direct call in case of stand-alone execution.

3.5. Graphic protocol

The graphic protocol therefore includes the following messages sent from the interface to the remote machine:

- INIT. To start object redraw.
- SELECT. To compute distance to object.
- TRANSLATE, SCALE, ROTATE. To modify data.

Conversely, the following messages are sent from the remote to the interface machine:

- SEND. To send data that need to be drawn.

- DISTANCE. To return distance information after a SELECT message.
- REDRAW. To force a graphic object redraw after the data has been modified.

3.6. Transparency from the user point of view

The proposed system was designed to be as transparent as possible from the user point of view, while remaining efficient. Indeed, most messages are completely hidden by the graphic system and the user programs call methods that cause the messages to be exchanged. In case of a stand-alone execution, the behavior of the system changes transparently for efficiency reasons. No messages are then exchanged.

The only difference with a stand-alone code from the user point of view is the need to split the `redraw` method in a "remote" part that sends data and a "local" part that retrieves data. Although message exchanges themselves are hidden in the get/send functions, two different methods are needed since only the "remote" part can read from the data object.

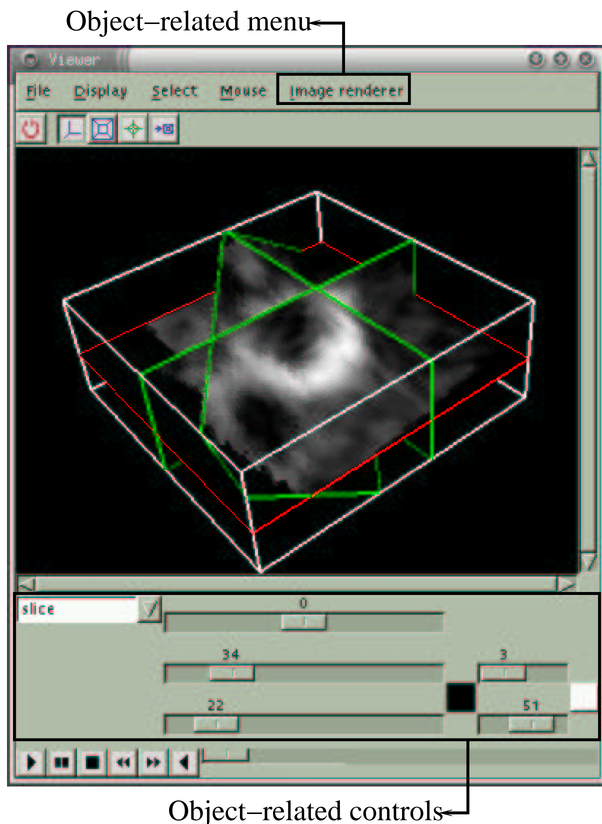

Figure 7. Object-related menu and controls on a 3D+T medical image renderer.

4. Results

This section presents preliminary results on a simple deformable contour object used for interactive segmentation of medical images. Figure 8 shows an interface similar to the one of figure 1 on which appears a graphic deformable contour overlaid on one slice out of a 3D carotid MRI. The contour and image data are stored on the remote machine. The window is grabbed from the local machine screen. The widgets at the bottom of the window are used to start and stop the segmentation algorithm.

5. Discussion

In this paper we presented a 3D object visualization interface for interacting with remote medical applications. The framework is generic and can indeed adapt to many different applications, even outside the medical field. The system is both efficient and transparent from the user point of view. It intends to ease the creation of medical image processing algorithms on emerging grid architectures.

Efficiency is needed to insure maximal performance in application for which interactivity may be critical (real time constraints may be needed). One purpose of grid technologies is to bring additional computing power to the user. The graphical system should therefore not compensate for the computational improvement.

It is mandatory that the system remains easy to use and as transparent as possible for the user in order to ease new

algorithm developments. Our system can run stand-alone when no remote server is available without code modification or recompilation. The message exchanges are hidden from the user.

Grid technologies already raised a large interest reflected by many research works on efficient scheduling, data management, and data transmission strategies. It will play a key role in a near future for processing large amounts of data using a limited set of shared resources. The new network generation with increased bandwidth, dedicated services, and guaranteed quality of services will boost management of large and distributed collections of data. This work proposes a software architecture to efficiently process, visualize, and interact with remote 3D medical images. It makes grid technologies available for interactive medical applications.

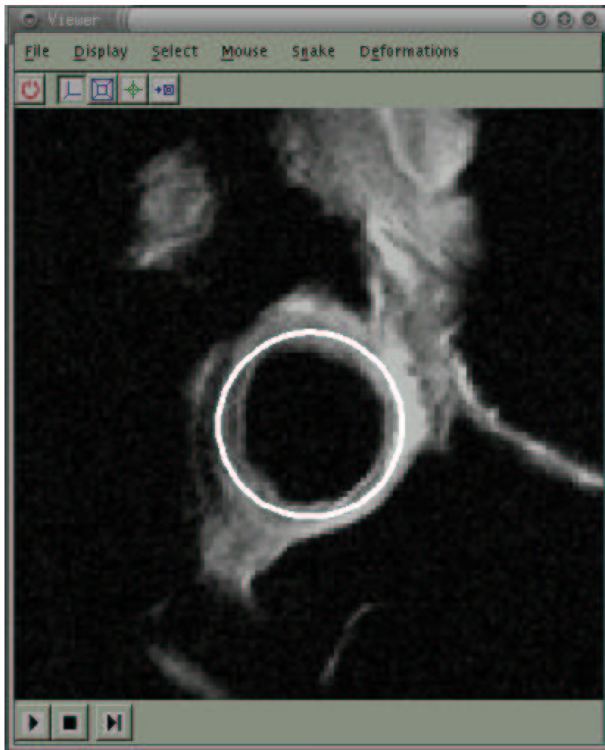

Figure 8. Deformable contour object overlaid on a carotid MRI slice. The data are lying on a remote machine and visualized on a local machine.

Acknowledgments

This work is partly supported by the European DataGrid project (`http://www.eu-datagrid.org/`).

References

[1] R. Acharya, R. Wasserman, J. Sevens, and C. Hinojosa. Biomedical Imaging Modalities: a Tutorial. *Computerized Medical Imaging and Graphics*, 19(1):3–25, 1995.

[2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *proceedings of the 8th IEEE Symposium on High Performance Distributed Computing*, 1999.

[3] C. Bajaj, V. Anupam, D. Schikore, and M. Schikore. Distributed and Collaborative Volume Visualization. *IEEE Computer*, 27(7):37–43, July 1994.

[4] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level Scheduling on Distributed Heterogeneous Networks. In *Supercomputing*, Pittsburgh, PA, USA, Nov. 1996.

[5] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *to appear in the Journal of Network and Computer Applications*, 2001.

[6] H. Delingette, E. Bardinet, D. Rey, J.-D. Lemarchal, J. Montagnat, S. Ourselin, A. Roche, D. Dormont, J. Yelnik, and N. Ayache. YAV++: a software platform for medical image processing and visualization. In *IEEE International Workshop on Model-Based 3D Image Analysis (IMVIA'01)*, Utrecht, The Netherlands, Oct. 2001.

[7] K. Engel, O. Sommer, and T. Ertl. A Framework for Interactive Hardware Accelerated Remote 3D-Visualization. In *Data Visualization*, pages 67–77. Springer, 2000.

[8] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.

[9] K. Kim. APIs for Real-Time Distributed Object Programming. *IEEE Computer*, 33(6):72–80, June 2000.

[10] W. Lefer and J.-M. Pierson. A Thin Client Archietcture for Data Visualization on the World Wide Web. In *proceedings of the International Conference on Visual Computing*, Goa, India, Feb. 1999.

[11] Object Management Group. Common Object Request Broker Architecture, Version 2.5. In *http://www.omg.org/*, Sept. 2001.

[12] C. Schmidt and F. Kuhns. An overview of the Real-Time CORBA Specification. *IEEE Computer*, 33(6):56–63, June 2000.