33rd Annual International Conference of the IEEE EMBS
Boston, Massachusetts USA, August 30 - September 3, 2011

MEDIFRAME – Remote Volume Rendering Visualization Framework

Roland Unterhinninghofen, Frederik Giesel, and Rüdiger Dillmann

Abstract— **Tablet computers, netbooks, and other mobile devices find their way into medical applications. However, advanced visualization such as volume rendering of tomographic data is too demanding for these devices. Hence the concept of remote visualization gains attention again. Using powerful servers views are rendered and transmitted as video-stream to the mobile devices in real-time. In this article we present a new extension to our software framework MEDIFRAME allowing easy setup of remote visualization in the medical imaging domain. We give an overview of the general visualization architecture and explain the remoting component in detail. Tests from different cities in Europe revealed good latency and rendering times as well as a surprisingly smooth user experience. We conclude that our remote visualization framework is a handy, functional extension to medical visualization applications.**

I. INTRODUCTION

Visualization is fundamental when working with medical imaging. It goes from as simple tasks as visualizing 2D tomographic images, over more sophisticated 3D volume rendering to complex scenes combining images with surface objects from segmentation, markers or glyphs. During the last decade a large number of medical visualization and image-processing toolkits and software have been developed. Besides commercial software from imaging equipment manufacturers and other companies, various freely available software exist for research purposes. Among the latter there are well-known tools such as 3D-Slicer [1], MevisLab [2], OsiriX [3], MITK [4], and our own MEDIFRAME [5]. These tools typically build upon or integrate toolkits like the Visualization Toolkit (VTK), the Insight Toolkit (ITK), or the Dicom Toolkit (DCMTK).

Usually the software runs on standard PC hardware; some more demanding applications extensively use parallel execution on multi-core CPUs or benefit from GPU acceleration. However, with recent tablet computers, smartphones, netbooks, and other light weight computers finding their way into medical application new challenges arise. There are in fact some dedicated applications, e.g. OsiriX for iPad showing 2D tomographic images. But although these devices are equipped with increasingly powerful processors and even 3D graphics capabilities, their computational power is still far from sufficient for advanced interactive medical visualization, especially when it comes to volume rendering. Hence the concept of remote visualization gains importance again.

With remote visualization powerful servers render views and transmit them as video-stream to a client computer in real-time. While this concept itself is not new, there is little dedicated to mobile devices in clinical applications. There are a number of solutions developed for general purpose scientific data visualization, e.g. the Chromium renderserver [6]. Also, some remote desktop solutions such as VirtualGL for Unix allow remote visualization with full OpenGL support [7]. The standard Microsoft Windows remote desktop, on the contrary, does not, as it involves a special graphics driver without GPU support.

However, none of the toolkits and tools commonly used in the medical image visualization and processing community provide first-class support for developing remote rendering applications. Consequently we present our approach to integrate a remote visualization API into our software framework MEDIFRAME.

II. METHODS

A. Overview

MEDIFRAME is a software framework providing high-level programming models for the development of medical applications particularly in the field of tomographic image processing and visualization. With its current, third generation it has moved away from pure C++ to a mixture of C#, C++/CLI, and native C++ based on the Microsoft .NET framework. It therefore benefits from the achievements of modern virtual machines such as reflection as well as from modern syntax. Also, it profits by the huge and mostly consistent libraries including string handling, files, XML, networking. Finally, with Windows Presentation Foundation (WPF) a GUI framework is available that may appear overkill for research applications but offers a handy programming model and appealing user interfaces.

For 3D visualization or data processing, however, MEDIFRAME relies on VTK. For performance reasons VTK classes are not simply wrapped in .NET classes. Instead a new architectural layer above VTK is created where classes typically integrate several VTK classes and hence represent discrete functional units. These classes are implemented in C++/CLI – the managed derivative of C++ – as it allows mixture of native and managed code at ease. However, care was taken that transitions from managed to native code and vice versa do not occur in extensive loops which might decrease performance considerably.

This work was not supported by any organization

R. Unterhinninghofen is with Institute for Anthropomatics, Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany roland.unterhinninghofen@kit.edu

F. Giesel is with Department of Radiology, University Hospital of Heidelberg, 69120 Heidelberg, Germany f.giesel@med.uni-heidelberg.de

R. Dillmann is with Institute for Anthropomatics, Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany ruediger.dillman@kit.edu

MEDIFRAME consists of a growing number of subframeworks. Two of them, namely the visualization framework and the remote framework are presented in the following.

B. Visualization Architecture

The visualization architecture aims to build a bridge between WPF and VTK. WPF on the one side is designed for graphical user interfaces. Unlike former GUI libraries it does not use GDI but relies on DirectX. The whole UI is represented as a tree of visual objects (so called *visuals*) composing various graphical primitives. Using templates and styles it separates appearance and business logic and gives ample control over the design. Typically WPF UIs are defined using the *eXtended Application Markup Language* (XAML), an XML-based language to instantiate hierarchical structures of classes and initialize their properties. WPF support for 3D visualization is currently rudimentary and does not appear appropriate to build a medical visualization framework on.

VTK on the other side is designed for (three dimensional) visualization of data such as volume images, meshes and geometric primitives based on OpenGL. It defines a virtual world and a camera model to view it. Datasets are processed through pipelines of filter instances that typically end in a pair of `vtkMapper`/`vtkActor` instances to visualize them.

One issue when integrating VTK with WPF is the so called *air space* problem: OpenGL and hence VTK is rendered into classic Win32-Windows which, however, may not overlap regions controlled by WPF. MEDIFRAME provides two solutions to this problem. The direct solution makes use of the WPF class `WindowsFormsHost` which hosts a Windows Forms control that in turn has a `vtkRenderWindow` as its child. The second, indirect solution stems from the remote visualization part: it uses VTK offscreen capabilities to render into memory. The resulting image is then displayed through standard WPF classes. While slightly slower the second solution has several advantages: (1) it avoids flickering when resizing the window, (2) it allows true overlapping of other WPF controls over the rendered image, (3) the rendered image acts as full WPF control and takes part in all transformation operations etc.

Fig. 1 gives an example how to define 2D and 3D scenes. `Viewport2D` and `Viewport3D` are the root instances which are WPF visuals and may hence be integrated into arbitrary WPF GUI. The `Frame2D` defines the slicing plane to be displayed (position and normal in world coordinates). Properties of child visuals such as `PlaneVisual` or `LabelVisual` are directly assigned in XAML. For the `ImageVisual` a concept called *binding* is used to assign the image data (`Source` property) and color transfer function (`ColorMapper` property). The data is taken from an object that is assigned to the `DataContext` property of the root object which is then inherited by all children. The desired source property is defined using the `Path` expression. Note that both `ImageVisual` objects bind the same image object; also note that the light blue plane is the same that controls `Frame2D`. In a real application these and more properties would be bound to a common model object to allow interactive dependencies between 2D and 3D views. Also templates would help to build common visualization scenarios. However, this topic is beyond the scope of this article.

C. Volume Rendering

The term volume rendering subsumes a number of techniques to render tomographic images three-dimensionally. In our context we use it synonymously to *volume ray casting* which is a volume rendering technique often applied to CT data. It basically computes rays of sight being continuously absorbed while travelling through the volume.

Since ray casting is a computationally demanding process it is particularly eligible for remote visualization where powerful server hardware performs the rendering and sends the image to light-weight clients. Because the ray casting algorithm is highly parallelizable, it is very often implemented on graphics hardware (GPU). In MEDIFRAME we integrated an own implementation based on GLSL (OpenGL Shading Language) which can be used in remote visualization [8].

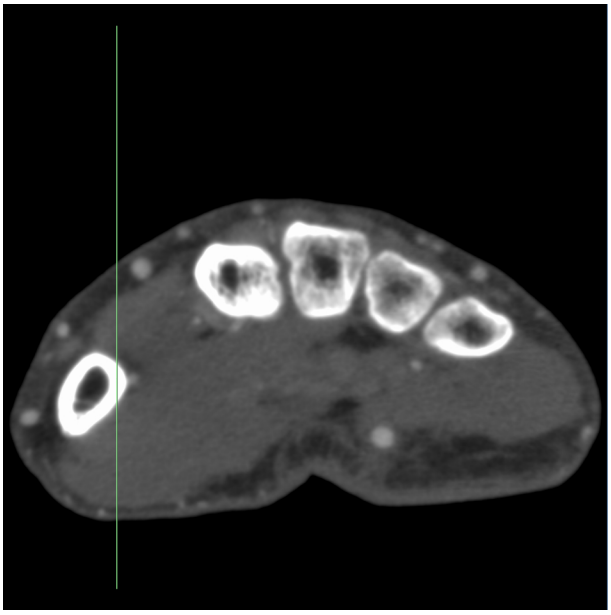
D. Remote Visualization

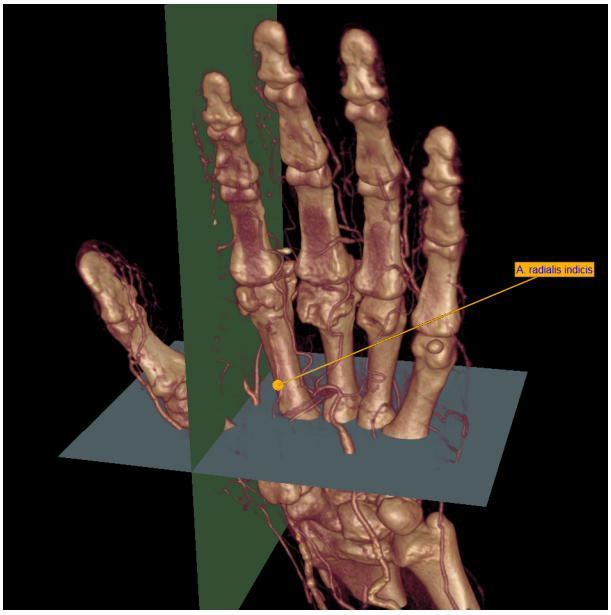
Remote visualization basically means rendering a view on a server and transmitting the resulting pixel image to a client computer. In view of application design the question arises what parts of an application window are remotely rendered. In the remote desktop scenario the application completely runs on the server and its window is transmitted as a whole. Alternatively, the basic GUI is rendered on the client-side and only the computationally expensive parts are rendered remotely – in our case complex 2D or 3D visualizations of medical image data. This approach resembles *Rich Internet Applications* (RIA) where GUI and its direct application logic is handled on the client side while complex data processing, e.g. using a database, is done on the server side.

MEDIFRAME deals with remote visualization on the viewport level, i.e. each `Viewport` instance is rendered and transmitted separately. It hence relies on the visualization architecture described above and additionally introduces the following concepts:

- A `Connection` object to handle a socket connection
- A `Server` to listen to incoming connections and to initiate a connection
- A `RemoteApplication` object to manage the whole application logic and to hold `Viewport` objects
- A `Session` object to store the current state

In a WPF desktop application a `Window` instance is typically the root object of the GUI tree. However, in the remote visualization scenario there is no GUI on the server side and the GUI tree reduces to a number of `Viewport` instances. They are registered directly with `RemoteApplication` that acts as root instance in a MEDIFRAME remote scenario. While properties of the visuals could be directly bound to properties in the `RemoteApplication`, an additional `Session` object is provided to bind these properties. The `Session` object serves as `DataContext` of the viewports






```
xmlns:scene="clr-namespace:Mediframe.Scene;
        assembly=Mediframe.Scene.Satellite"

<scene:Viewport2D>
 <scene:Frame2D Plane="0,0,80,0,0,1">
  <scene:ImageVisual Source="Binding Path=Img"
      ColorMapper="Binding Path=CM2D"
      InterpolationMode="Linear" />
  <scene:PlaneVisual Plane="100,0,0,1,0,0"
      SurfaceColor="LightGreen"/>
 </scene:Frame2D>
</scene:Viewport2D>
```

```
<scene:Viewport3D>
 <scene:Frame3D>
  <scene:ImageVisual Source="Binding Path=Img"
      ColorMapper="Binding Path=CM3D"
      Raycaster="GLSL_2Pass"/>
  <scene:PlaneVisual Plane="0,0,80,0,0,1"
      SurfaceColor="LightBlue"/>
  <scene:PlaneVisual Plane="100,0,0,1,0,0"
      SurfaceColor="LightGreen"/>
  <scene:LabelVisual AnchorPosition="86,57,90"
      Position="800,600" LeaderColor="Orange"
      Foreground="Blue" Background="Orange"
      Padding="2" FontSize="16"
      Text="A. radialis indicis"/>
 </scene:Frame3D>
</scene:Viewport3D>
```

Fig. 1. Example of scenes decribed in XAML. Left: 2D view, Right: 3D view. Note dependencies between both views.

and hence, by simply swapping the `DataContext`, multiple sessions, e.g. from different users, may share the same `RemoteApplication` and `Viewport` instances. This is particularly useful when a large number of users would outrun the server's memory capacities.

The programming model to set up a remote application server in MEDIFRAME is hence fairly simple:

1) Define scene tree and register with `RemoteApplication`. This can be done in a single XAML file.
2) Define session class and bind properties to visuals.
3) Setup server adress

In earlier versions we used H.264 or WMV1 video streams to encode and transmit the rendered views [9]. However, the encoding is time-consuming and introduces synchronization problems. We finally found it advantageous to encode and transmit each frame as separate JPEG image. While this slightly increases the network load it significantly reduces complexity and gives better control over image quality in static and dynamic, i.e. interactive, display. With the `Viewport` class inherently producing a pixel image through VTK/OpenGL offscreen rendering, encoding is performed through .NET JPEG-Encoder. The resulting data is sent to the client over the socket connection.

When a client connects to the server, a new `RemoteApplication` instance is created and a new socket connection is established. Each `Viewport` is rendered initially and sent to the client. When the client sends mouse and keyboard events to the server, they are routed to the respective viewport where they are processed. The rendering itself, however, is not started immediately when an event has been processed. Instead there is a rendering cycle with a period of typically 40 ms that synchronously renders invalidated views. This is to reduce computational and network load and to avoid events to queue which would make the rendering to lag behind.

The client is currently implemented as Microsoft Silverlight application. Each viewport that is defined in the remote application is represented by a dedicated control based on the `Canvas` control. When the `Canvas` is resized the client sends the new size to the server. The server, however, does not necessarily render the image in the desired size; it may use other criteria – e.g. server load or algorithms optimized for specific sizes – to determine the actual size.

TABLE I

ROUND TRIP TIMES FROM DIFFERENT LOCATIONS IN EUROPE.

Location	Distance	Network	Time
Karlsruhe	200 km	WAN	71 ± 35 ms
Heidelberg	175 km	WAN	74 ± 39 ms
Munich	150 km	DSL	92 ± 48 ms
Siegen	270 km	DSL + WLAN	141 ± 58 ms
Basel	330 km	DSL + WLAN	136 ± 53 ms
Malta	1530 km	DSL + WLAN	487 ± 82 ms

Hence, the client potentially needs to zoom the received image to fit into the canvas. Since (mouse) coordinates on the client and on the server side are consequently not the same, they are normalized before transmission.

III. RESULTS

For evaluation purposes we set up an application that displays a CT dataset and a numbers of labels (similar to the example above). It was installed on a dedicated server running Microsoft Windows Web Server 2008 R2, equipped with one Intel Xeon W3565 CPU (4 Cores, 8 Threads, 3.2 GHz), 16 GB RAM, and two GeForce GTX 480 graphics boards. The server was housed in a data center close to the city of Nuremberg, Germany.

On the client side we used a notebook (Intel Core2 T7200 CPU and Nvidia Geforce Go 7400 graphics) running Microsoft Windows 7 at 1280 x 800 pixels resolution.

Two aspects determine speed and responsiveness of the remote visualization. The first is the time to render the view and to encode the resulting image as JPEG. The second is the time to transmit user events from client to server and to transmit the image frames from server to client.

Rendering and encoding time depends on image size and rendering quality which is dynamically adapted to obtain interactive rates. As mentioned above rendering is clocked with 40 ms. A dataset of $512 \times 512 \times 400$ voxels was rendered smoothly without considerably degrading image quality.

Network times depend, among other factors, on the network adapters used, on the distance between client and server, and the number of routers in the line. We did, however, not evaluate each of these factors separately but measured the entire round trip time, i.e. the time to send a mouse event to the server, to render and encode the image, and finally to send it to the client. These measurements were performed from different locations in Europe using either WAN or DSL connections, the latter partly combined with WLAN as it is often found at home or in public places such as hotels. Mean time and standard deviation were computed from 100 measurements per location. The results are shown in table I. Given distances are air-line distances to Nuremberg, Germany, where the server was housed.

Round trip times obviously increase with longer distances. Also DSL connections tend to introduce higher variations. However, even with rather high values for Malta (487 ± 82 ms) the user experience was still close to acceptable.

IV. CONCLUSIONS AND FUTURE WORKS

A. Conclusions

We presented a remote visualization framework dedicated to medical visualization and particularly appropriate for remote raycasting of tomographic data. While the basic principle is not new, our solution offers a comprehensible and easy-to-use API entirely integrated into MEDIFRAME. It benefits from its interface to the well established VTK and from the powerful .NET-Framework. Also, applications can be developed so that most parts can be identically used in local and in remote MEDIFRAME applications. However, the usability of the API needs to prove in practice.

Results reveal a very good responsiveness – qualitatively as well as quantitatively. Adaptive rendering effectively reduces processing and transmission times.

B. Future Works

Our solution has a number of limitations to be adressed in future. First of all it does not yet exploit the power of multi-GPU systems because GLSL is limited to the primary GPU. To overcome this limitation a new CUDA implementation of the raycasting algorithm is being developed. It will allow multiple clients to access the server synchronously and have it render different views at the same time.

To further speed up the system, image compression could be implemented on the GPU as-well. However, loss in image quality from compression and intermediate resizing steps needs to be minimized.

Finally it will be interesting to further optimize server-side computation and adaptation of rendering quality. Also different network conditions, including inter-continental connections, and different client computers will be evaluated.

REFERENCES

[1] S. Pieper, B. Lorensen, W. Schroeder, R. Kikinis, "The NA-MIC Kit: ITK, VTK, Pipelines, Grids and 3D Slicer as an Open Platform for the Medical Image Computing Community" *Proceedings of the 3rd IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, 2006, vol. 1, pp. 698–701

[2] M. König, W. Spindler, J. Rexilius, J. Jomier, F. Link, and H.O. Peitgen, "Embedding VTK and ITK into a visual programming and rapid prototyping platform", *SPIE Medical Imaging*, 2006.

[3] A. Rosset, L. Spadola, and O. Ratib, "OsiriX: An Open-Source Software for Navigating in Multidimensional DICOM Images", *J. Digit. Imaging.*, 2004 vol. 17(3), pp. 205-216.

[4] I. Wolf, M. Vetter, I. Wegner, T. Böttger, M. Nolden, M. Schöbinger, M. Hastenteufel, T. Kunert, H.P. Meinzer, "The medical imaging interaction toolkit", *Med. Image. Anal.*, vol. 9(6), 2005, pp. 594–604.

[5] S. Seifert, R. Kussaether, W. Henrich, N. Voelzow, R. Dillmann, "Integrating Simulation Framework MEDIFRAME", *Proceedings of the 25 IEEE Engineering in Medicine and Biology Conference*, 2003, pp. 1327-1330

[6] B. Paul, S. Ahern, E. Wes Bethel, E. Brugger, R. Cook, J. Daniel, K. Lewis, J. Owen, and D. Southard, "Chromium Renderserver: Scalable and Open Remote Rendering Infrastructure", *IEEE Tran. Vis. Comput. Graphics*, vol. 14(3), 2008, pp 627–639.

[7] D.R. Commander, "VirtualGL: 3D Without Boundaries – The VirtualGL Project", *http://virtualgl.sourceforge.net/*, 2007.

[8] S. Suwelack, E. Heitz, R. Unterhinninghofen, and R. Dillmann, "Adaptive GPU Ray Casting Based on Spectral Analysis", *Proceedings of Medical Imaging and Augmented Reality (MIAR)*, 2010, pp. 169-178

[9] S. Suwelack, S. Maier, R. Unterhinninghofen, and R. Dillmann, "Web-based interactive volume rendering", *Stud. Health. Technol. Inform.*, 2011, vol. 163, pp. 635–637