Remote Rendering of Industrial HMI Applications

Pablo Gómez Pérez
Institute of Applied Knowledge Processing
Johannes Kepler University
Linz, Austria
e-mail *pablo.gomez@faw.jku.at*

Wolfgang Beer / Bernhard Dorninger
Software Competence Center Hagenberg
Hagenberg, Austria
e-mail *wolfgang.beer/bernhard.dorninger@scch.at*

***Abstract*—Remote access to plant/machine HMIs has always been an area of great interest. A broad range of well-established solutions is available to fulfill the various tasks from remote operation to remote maintenance (e.g., VNC). Depending on the regarded solution, there are more or less strong demands on the respective client device. In the course of the gaining popularity of cloud computing-especially the gaming on demand sector-a potentially attractive alternative arises with remote rendering of applications based on video streaming. In our paper we provide insights in our efforts of developing a first prototype of a such a video streaming based solution and discuss the suitability of the approach in an industrial context.**

***Index Terms*—remote rendering, remote maintenance, remote access, HMI applications, video encoding, streaming**

I. Introduction

Industrial machine control systems and networked, interconnected industrial plant and facility automation systems may involve a large amount of realtime digital control units as well as human machine interfaces (HMI). The processing power of such digital systems range from low-level 8 bit micro controller units, over programmable logic controls (PLCs) to full scale 64-bit servers that handle the connection to a company's enterprise resource planning (ERP) and production planning systems (PPS). Production machines (e.g., plastic moulding) usually heavily depend on configurations, parameters and operational decisions, as well as on error handling, all of which is typically covered by human operators. Such a machine is usually equipped with its own HMI, which allows an operator to supervise, configure and control the production process. In former times, such HMIs were barely composed of a set of control buttons and switches combined with some lamps for visual feedback. Today, quite complex HMIs consisting of some sort of tangible user interface and a graphical user interface (GUI) with respective input devices are a frequent sight. In addition, touchscreens became a common replacement for the monitor/keyboard/mouse combination over the last decade. With the broad emergence of multi-touch capable screens in consumer devices, such as smartphones and tablets, solutions based on multi-touch technology are becoming increasingly popular in industrial applications, too. Due to this and considering the fact that GUI software has become increasingly complex over the years involving effects and animations, demands concerning graphics hardware have grown as well. To render a user interface onto such a mounted touchscreen, the machine has to include an extra CPU (or at least share a CPU with the PLC), a high end graphics card with 3D acceleration support and enough memory to render to a given screen resolution. Equipping each machine with its own private GUI terminal has thus become a non negligible cost factor, especially with smaller production machines in a vendor company's low price segment (relative cost of GUI related to cost of whole machine). This weighs even more considering that large plants may consist of dozens to hundreds of similar production machines.

Furthermore, there is the noticeable fact that in a high number of cases these GUI solutions are seldom used during operation. In highly automated production environments, production machines are once set up before running continuously for a longer period of time. Typically, one machine operator is responsible for a large number of machines, but only needs to interact when the production process has to be altered or machine faults occur. This implies that a locally available but also extensive GUI solution might be a waste of resources. Of course there are special solutions for large plants like SCADA/guidance systems or Manufacturing Execution Systems (MES), but usually these systems do not cover the rarely needed details of a machine rather than a broad overview. In addition, these solutions are usually located in a control stand. Operating and parametrizing a machine as well as fault diagnosis and rectification however generally necessitates a locally available GUI. Thus a solution providing a locally available, extensive GUI, but at the same time sparing expensive hardware would be a considerable option. Remote Rendering is a promising approach which might fulfill exactly these demands.

In this work we want to show that in a modern industrial production facility, where all machines are reliably interconnected by high bandwidth networks, locally rendered GUIs could be replaced by a remote rendering approach. We hope to reduce the costs for a single machine, to save energy by

reducing the need for high-end graphical rendering on local machines and to provide the GUI application's functionality on various devices for human operators.

The remainder of this paper is organized as follows: Section II provides a brief overview of efforts in remote rendering and cites relevant related work. In section III, we outline our approach. Selected aspects thereof are described in more detail in section IV. Section V then highlights some interesting performance issues. The following Section VI states the most important of our open issues and proposes some of our further work. Finally, section VII provides a short conclusion.

II. Related Work

Rendering and displaying applications on remote computers is nothing new. The principle of remote rendering has already been implemented with the first versions of Pixar's rendering software *Renderman* back in the 1980s [1]. Remote maintenance systems have been common in industrial environments for quite a long time now. There are classical *desktop sharing* applications to transmit the desktop of a computer to a remote host, the most prominent one being VNC [2] and its numerous implementations. *Terminal Services* allow the private use of a remote host. Unix/X11 and related operating systems have always been capable of providing a private terminal to a number of users, Microsoft Windows provides terminal services since Windows 2000. More recently, *Application Virtualization* gained popularity. Software applications need not to be installed on a local computer but are rather transmitted over the network and then executed locally or even executed on a centralized application server with the GUI being transmitted to a remote client. For instance, Citrix's XenApp [3] is such a product providing both ways of application virtualization.

However, all the mentioned solutions are often proprietary and require costly licenses and/or are targeted at a specific operating system. In any case, the client device is responsible for fully rendering the UI which requires reasonable graphics hardware. Solutions employing bitmap based protocols like VNC are often not suitable for fluid, highly animated applications. On low performance systems, users often experience juddering and flickering applications.

An interesting alternative emerged from the field of online gaming. In [4], Trzuya et al. describe their approach of a platform for ubiquitous gaming and multimedia. Although initially focusing on transmitting graphics commands [5] , the platform also features an alternative for more underperforming devices employing the transmission of game output via video stream [6]. This is of even more interest since modern low cost devices (like e.g., the Raspberry Pi [7]) tend to have low performance graphics hardware but on the other hand feature hardware media decoding support. The idea already has been adopted by traditional remote desktop approaches. In [8], Simoens et al. describe an extension to VNC's Remote Frame Buffer (RFB) protocol, which allows to transmit video within RFB messages.

Recently, the video streaming based approach gained broader attention due to provisioning remotely rendered games in the cloud (Barboza et al. [9], Zhao et al.[10]). Meanwhile, there is a number of commercial online gaming platforms providing their services on base of remote rendering technologies (e.g., OnLive[11], Otoy[12]).

III. Approach

A. Goals

The purpose and contribution of this work is to discuss a remote rendering approach involving video streaming for replacing local rendered industrial machine GUIs. By practically implementing this approach, the GUI applications are no longer executed on the production machine rather than on a shared rendering server. As the host should be a capable, high end server device, it will be able to execute and render a reasonable number of GUI applications. Therefore, the production machine no longer has its own private GUI, which in turn allows to spare the respective hardware.

In detail, remote application rendering means to render individual industrial GUI on a powerful 3D accelerated server machine, to record and video encode the rendered application and to constantly stream the encoded video to a listening client. The client could be a touchscreen mounted on a machine, within a control room or even mobile devices, such as Tablet PCs. The workload on the client is reduced to displaying an encoded video stream and to collect the user feedback, such as touch events, in order to send the user input back to the rendering server.

Our particular goals are:

1) Execute and render multiple machine HMI applications on a central server.
2) Record each application's output and stream it to the client using an efficient video format
3) Allow a broad range of application devices to display application output. Especially low cost devices with built in video decoding support, like the Raspberry Pi [7], shall be targeted.
4) The streamed application must behave similar to a locally executed application, i.e. the platform should provide a satisfactory user experience.

In the next section, we will describe the steps to fulfill the specified goals.

B. Conceptual Overview

Our concept of remote rendering consists basically of two main sequences, which can be further divided into more detailed steps (see Figure 1).

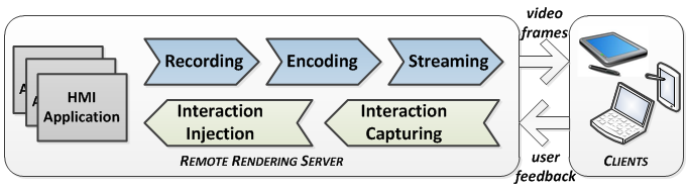

Figure 1. Conceptual Overview

Firstly, there is the upstream sequence which involves the following steps:

- Initialize and start the appropriate application
- Record the running application instance
- Encode the recorded application images to a video
- Stream the video over a network

After fulfilling these steps, the client may connect to the video stream and the user may watch the application output. But of course, user interaction must be captured and sent back to this application. This is considered the *downstream sequence*, which consists of two more steps.

- Capture user interaction
- Inject user interaction into correct application

Providing a reasonable level of user experience requires a considerable frame-rate, which allows a client to display a smooth and flicker-free video. The upstream sequence has to utilize efficient methods of recording and encoding. Employing the rendering server's graphics hardware for these steps seems to be a promising approach. Naturally, also the downstream sequence needs to be efficient. The higher delays and latency times become, the more inaccurate application control will get. Under normal circumstances, i.e. when regarding locally executed interactive applications, users consider latencies of less than 0.1 seconds as acceptable for actions deemed as instantaneous [13]. This value is also demanded in our case.

C. Topology / Architectural Overview

The prototype's setup resembles a multi-tier topology, which is depicted in Figure 2.

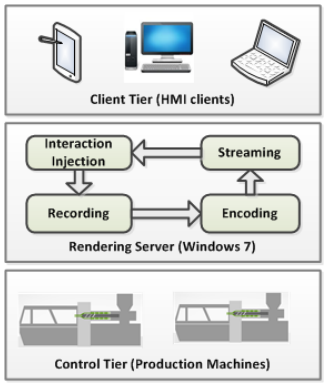

Figure 2. Architectural Overview

The production machines host the control system (PLC) and provide the functional interface for the HMI application. There is no difference if the HMI application would run locally on the machine's PLC or on an active built-in terminal. Of course, this assumes that HMI and PLC employ a network based communication. In our case, the HMI and the PLC use remote procedure calls to communicate.

The middle tier is formed by the rendering server. It provides an interface for clients allowing them to select and start a HMI application instance for any managed production machine. Since it will be responsible for recording, encoding and streaming multiple application instances, it must be an adequate powerful server. In addition the server has to receive user feedback and inject the user interactions into the correct running application instance.

The client side is responsible for decoding and displaying the video as well as for capturing the user interactions. As pointed out, it is not necessary to provide really powerful graphics hardware, but of course should have hardware support for video decoding.

IV. Implementation

The first step in our work was the implementation of a basic server prototype being capable of rendering at least a small number of HMI applications and correctly capturing the respective client feedback. For the server environment we have chosen Microsoft Windows 7 while on the clients side we experimented with Android as well as common desktop PCs with both Linux and Microsoft Windows. The following subsections cover a closer look the steps described in the conceptual overview in the context of our prototype.

A. Upstream sequence

The upstream sequence covers the tasks recording, encoding and streaming with the encoding being further refined into more detailed steps. Figure 3 shows the steps and components involved. In the following subsections we will describe each task and its steps in a little more detail.

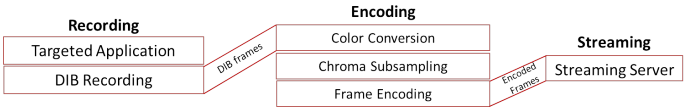

Figure 3. Server tasks and steps

1) Recording: Whenever a HMI application is started it is rendered on the server's graphics hardware. This output must be captured and converted to a video. As high performance is required, we ought to work close to the hardware and the operating system. Since have based our prototype on Windows 7, we use the Windows GDI API, which allows to record any window content on the desktop to device independent bitmaps (DIB) [14]. In addition, we try to use the Windows 7 Aero Glass feature, which allows to record window output from applications which are not or only partially visible. However, working with bitmaps is not suitable for our purpose, since video streaming has its own formats. Thus, the recorded bitmaps need to be encoded as a video.

2) Encoding: In this step the recorded frames are converted into the targeted video format, which is H.264 in our case. The main reason for our selection of H.264 was mainly due to its current popularity and the easy availability of codec software. In addition, the broad hardware support of this format was another benefit. However, encoding is a key task in our scenario, it should be handled as efficient as possible. In this context, the question arose whether usage of graphics hardware would lead to a significant performance edge. Thus, we have decided to try two different alternatives:

- GPU encoding based on the Nvidia Encoder *NVEnc* [15]. The *NVEnc* generates the video frame unit by using the graphics hardware. Nvidia provides its own proprietary

computing architecture (CUDA), so the NVEnc only works with Nvidia graphic cards. Newer Nvidia graphic adapters also feature hardware supported video encoding, which has been promised to be four times faster than the CUDA variant, while being more efficient in terms of power consumption.

- CPU encoding based on the library *x264* [16]. It is a free software library that forms the core of the library *libavcodec* - part of the multimedia framework *ffmpeg* and thus many web video services such as Youtube or Facebook. It works solely based on CPU algorithms.

Both methods produce *Network Abstraction Layer (NAL)* units, which are the building blocks of a H.264 video stream. The encoder alternatives expect the input frames in the color space format YV12, a derivate of the YCbCr color space, which is the standard color space for use in videos. YCbCr is an encoding format originally developed for television, since it fits more to human vision than RGB. YV12 is a variant of YCbCr, which has a different structure making it more convenient for compression. Thus, we need to preprocess the recorded DIB frames, which are encoded in the RGB color space:

1) Color conversion: In a first step the frames have to be translated from RGB to the YCbCr color space. The YCbCr (luma, blue difference chroma, red difference chroma). Before doing the conversion from a DIB to a YCbCr frame we need to take into account the orientation of the captured frame, because, DIB frames typically have a bottom up representation and video frames usually are top down. After correcting the orientation the frames can be converted to the YCbCr color space, which follows a standardized algorithm (see ITU.BT-601 [17]).
2) Chroma subsampling. In a subsequent step these frames have to be sub-sampled to the YV12 standard. This means to maintain the luminance of YCbCr but sub-sampling the chroma Cb and Cr per square pixels. The different luma and chroma aspects are grouped together, which makes this type of encoding much more suitable for compression. However, since human vision has poor acuity to color detail (compared to that of luminance), there is no noticeable difference after the sub-sampling from the original color space. However, space is saved by reducing the number of bytes per pixel from 3 bytes to 1.5 bytes [18]. The resultant frame should look like this:

$$Y_1 ... Y_n Cr_1 ... Cr_{\frac{n}{4}} Cb_1 ... Cb_{\frac{n}{4}} \mid \ n \ mod \ 4 = 0$$

Regarding the implementation for the color conversion and the color subsampling, we have tried two different alternatives:

- *OpenCV + Sequential Algorithm:* We have used the implemented color conversion in the OpenCV library v2.4 [19] which provides two different approaches to do the color conversion. But OpenCV does not offer a chroma sub-sampling implementation, so we had to implement this ourselves.
- *FFmpeg:* is a powerful and well known multimedia framework that provides tools and developer libraries. FFmpeg has the capability of performing the necessary color conversion and chroma sub-sampling.

FFmpeg has shown to be more efficient for this task, especially regarding memory management. An alternative could be to develop a similar algorithm to the one implemented in FFmpeg, but with explicit GPU support.

After the preprocessing is done, the frame encoding itself can be performed, which is merely a call to the respective library. An interesting option offered by encoders is the possibility to downscale the recorded bitmaps. With this feature the rendering server may adapt the resolution of the streamed application to the client's capabilities.

3) Streaming: Once having a sequence of NAL packets, one needs an appropriate software component to stream those packets over the network. A powerful, freely distributable (LGPL) streaming server is available through the Live555 Streaming Media libraries [20]. Live555 Streaming Media is a set of C++ libraries that support video streaming standards such as the Real Time Streaming Protocol (RTSP) [21], which is the protocol of choice in our prototype. RTSP is a client-server multimedia presentation control protocol, designed to address the needs for efficient delivery of streamed multimedia over IP networks. The NAL packets are sent by the RTSP protocol which is layered atop of UDP. As UDP does not guarantee the order of the packages, RTSP mandates to establish a time stamp when the frames should be displayed. This measure allows to avoid potential problems such as decoding and displaying frames in a wrong order. Because of the fact that we are working with a live video source (i.e. our HMI applications), the time stamp of a frame is its generation time and its in our hands to ensure the correct frame order.

Live 555 offers two modes to stream to clients. Multicast allows to serve multiple clients at once, which is a suitable method for services like video on demand. In our case, Unicast is the better choice due to the fact we are aiming for private peer to peer connection scenario.

B. Downstream sequence

1) Interaction Capturing: Of course, it is not sufficient for our solution to just display the application. We also need to gather user interactions and forward them to the correct application on the rendering server. This could be either achieved by providing a specifically modified video player, which collects the user response or a small separate application doing this for us. For transmitting the interactions to the server we chose TUIO [22], a protocol developed for capturing and transmitting interactions from tangible multi-touch surfaces [23]. On a client device, a TUIO tracker is responsible to collect interaction events, marshaling them and sending them to a number of TUIO clients. So the client device acts as a TUIO server, whilst the rendering server is a client of this TUIO enabled device listening for user interaction messages. There are several TUIO tracker implementations available for free on various platforms.

2) Interaction Injection: We have to make sure that the feedback of a client device is mapped to the correct HMI

application on the rendering server. The most straightforward way to achieve is to have one TUIO client process per executed HMI application instance. This defacto remote control could also be included in the application in case the source code of the application is available. Another way is to inject the events with the help of a so called TUIO input bridge: the TUIO messages are converted to the respective Windows UI events and directly injected into the applications event queue.

TUIO has been primarily designed for touch surface applications, so - unlike in remote desktop protocols like RFB or RDP - there are no specific messages for mouse or keyboard interactions. However, TUIO offers the definition of custom messages, which may be used for this purpose.

One potentially negative issue may be the lack of reliable timing information. As TUIO relies on UDP too, loss of packages or wrong package order may occur. This of course may cause problems with interaction handling. However, with the coming version of TUIO (2.0), a time stamp is added for each message providing fine-grained timing information that will mitigate this issue.

V. Performance Issues

Because performance is the most important requirement, we want to highlight some selected aspects of our prototype. High performance of all steps is a crucial factor for providing a high level of user experience. For a smooth video replay we need at least 25 Frames Per Second streamed to the clients. In addition, latency of video replay should be as low as possible and application responsiveness as high as possible. On the server side, scalability is an important issue: How many applications may be executed, encoded and streamed on one system? This section covers the performance examination of different variants regarding the recording and encoding combination phases. We have measured the performance on two different systems, the first one being a desktop computer and the second one being a powerful graphics workstation.

Desktop CPU: Core i7 - 2600 @3.4GHz - 4 Cores, 8 GB RAM
GPU: Nvidia Quadro 600 (Fillrate: 10.2 GP/s)

Workstation CPU: 2xIntel Xeon X5690 @3.4Ghz - 6 Cores (total 12 Cores), 24 GB RAM
GPU: 2xNvidia GTX 580 (Fillrate: 2 x 37.06 GP/s)

Figure 4 shows a series of tests over the Recording task aiming to test the scalability when multiple applications are recorded. All recorded applications have the same resolution 1600x1200.The results also show the Aero Glass Feature decreasing the frame rate on both the Desktop and the even more powerful Workstation beyond the constraint of 25fps. Scalability, even on a high performance server is limited to one up to three applications recorded simultaneously, which would be unacceptable.

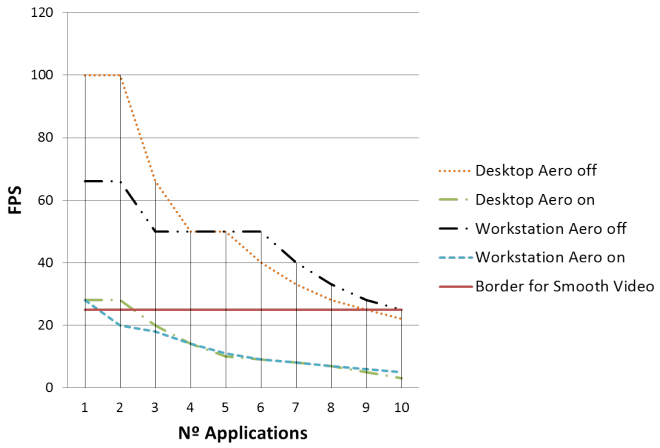

Figure 4. Recording

Figure 5 aims to show the scalability of our prototype regarding the recording *and* encoding multiple applications. All recorded and encoded applications have the same resolution 1600x1200. The encoding has been performed with the two analyzed video encoders: the GPU based NVEnc and the CPU based x264.

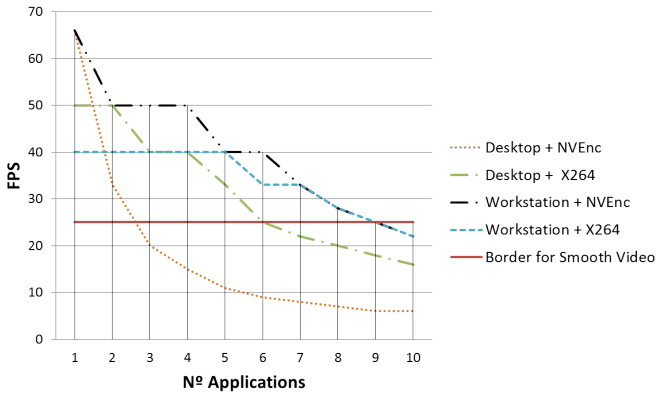

Figure 5. Recording + Encoding

It can be seen that the system *Desktop* does not have enough GPU power to allow a satisfying use of the GPU based encoder. In contrast, the x264 based encoding is more robust in terms of keeping up the frame rate. The tests on *Workstation* show both the GPU based and the CPU based encoding being significantly faster in all cases, so the selected approach definitly scales. However, when running 9-10 applications one can see the bottleneck from the recording task making its presence felt.

VI. Open Issues and Further Work

As this is work in progress, there are still many issues to deal with. Below is a list with the most important challenges which remain to be tackled in the near to middle future.

As shown in section V, there are still issues with recording the applications. Recording scales too low, even if using a high performance machine. Therefore we have to think of different recording methods than the ones employed in our prototype.

In addition, the current implementation causes a significant latency when transmitting the video of around 500ms, making the user's interaction with the application quite uncomfortable. One main reason is that it is usual to buffer videos on the client side to provide a continuous replay. This is especially necessary in environments with an unpredictable connection and network quality. Assuming we have a powerful and reliable production network, we might neglect connection issues. Thus we might keep buffer sizes down, since a larger buffer size in turn increases the playback latency. But buffer size is not the only influence on latency. Naturally the latency decreases considerably with lower resolutions. We assume the source of the remaining latency being mainly the streaming server and its configuration.

Another issue we plan to take care of in the near future is server integration and scalability. Currently, the prototype is made up from a number of different processes which are loosely coupled (e.g., data interchange via loopback interface) per application session. In the future, we plan a tighter integration here. This is especially the case for the server responsibilities concerning application start requests as well as interaction mapping and injection. We believe tighter integration may also decrease video latency.

VII. Conclusion

In this paper we presented our efforts regarding remote rendering of applications in an industrial context. We described the idea and approach of our work. Instead directly on the machine, HMI applications are executed on a central server, which streams the HMI application output as a video on demand. User interactions are captured on the client's device and are sent back to the server. We described the conceptual steps in our approach (recording, encoding, streaming - interaction capturing and injection) and selected aspects of their prototypical implementation. In addition, we compared the prototype performance on different target systems using different recording and encoding methods. Our measurements suggest an acceptable frame rate for generating a smooth video, albeit not with the top quality resolutions (e.g. Full HD). However, our preferred recording method didn't prove to be scalable enough. In addition, we still experience a high latency of the video playback. We assume that its roots reside within the streaming task. As pointed out, this is an issue we are working on at the moment.

Considering these limitations, our prototype is not yet ready for productive operation. Still we expect the approach of remote rendering via video stream has a future. This is demonstrated by the Online Gaming community and other professional solutions. What works for games and other highly interactive desktop applications, will also work for industrial HMI Applications. The latter will also be executed in the cloud and will be controllable from simple client devices with video rendering capabilities. This will help to reduce both deployment and maintenance costs, especially in large scale scenarios, i.e. plants with a higher number of machines.

References

[1] A. A. Apodaca and M. Mantle, "Renderman: Pursuing the future of graphics," vol. 10, no. 4. IEEE, 1990, pp. 44–49.

[2] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper, "Virtual network computing," *Internet Computing, IEEE*, vol. 2, no. 1, pp. 33–38, 1998.

[3] (2012, 12) On-demand application delivery. Web. Citrix Systems Inc. Accessed 13.12.2012. [Online]. Available: http://www.citrix.com/products/xenapp/overview.html

[4] Y. Tzruya, A. Shani, F. Bellotti, and A. Jurgelionis, "Games@large-a new platform for ubiquitous gaming and multimedia," *Proc. BBEurope, Geneva, Switzerland*, pp. 11–14, 2006.

[5] P. Fechteler and P. Eisert, "Remote rendering of computer games," Fraunhofer Institute for Telecommunications, Tech. Rep., 2007.

[6] A. Laikari, P. Fechteler, P. Eisert, A. Jurgelionis, F. Belloti, and A. D. Gloria, "Games@large distributed gaming system," VTT Technical Research Centre of Finland, Fraunhofer Institute for Telecommunications, University of Genoa, Tech. Rep., 2009.

[7] (2012) Raspberry pi: An arm gnu-linux box for usd25. WWW. Raspberry Pi Foundation. Last visited: 20130215. [Online]. Available: http://www.raspberrypi.org/

[8] P. Simoens, P. Praet, B. Vankeirsbilck, J. De Wachter, L. Deboosere, F. De Turck, B. Dhoedt, and P. Demeester, "Design and implementation of a hybrid remote display protocol to optimize multimedia experience on thin client devices," Dec. 2008, pp. 391–396. [Online]. Available: http://dx.doi.org/10.1109/atnac.2008.4783356

[9] D. C. Barboza, H. L. Junior, E. W. G. Clua, and V. E. F. Rebello, "A simple architecture for digital games on demand using low performance resources under a cloud computing paradigm," in *Proceedings of the 2010 Brazilian Symposium on Games and Digital Entertainment*, ser. SBGAMES '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 33–39. [Online]. Available: http://dx.doi.org/10.1109/SBGAMES.2010.34

[10] Z. Zhao, K. Hwang, and J. Villeta, "Game cloud design with virtualized cpu/gpu servers and initial performance results," in *Proceedings of the 3rd workshop on Scientific Cloud Computing Date*, ser. ScienceCloud '12. New York, NY, USA: ACM, 2012, pp. 23–30. [Online]. Available: http://doi.acm.org/10.1145/2287036.2287042

[11] (2012) Play on-demand video games over the internet. WWW. OnLive. Last visited: 20130218. [Online]. Available: http://www.onlive.com/

[12] (2012) Streaming games, movies and applications to any device. Otoy Inc. Last visited: 20130218. [Online]. Available: http://www.otoy.com

[13] J. Nielsen and J. T. Hackos, *Usability engineering*. Academic press San Diego, 1993, vol. 125184069.

[14] R. Gery, "Dibs and their use," *Microsoft Developer Network Technology Group, http://msdn.microsoft.com/en-us/library/ms969901.aspx*, 1992, Accessed October 12, 2012.

[15] NVIDIA. (2012, Accessed November 11, 2012) Nvidia cuda video encoder. NVIDIA Corporation. [Online]. Available: http://docs.nvidia.com/cuda/samples/3_Imaging/cudaEncode/doc/nvcuvenc.pdf

[16] (2012) Videolan - x264 library. WWW. VideoLAN Organization. Last visited: 20130220. [Online]. Available: http://www.videolan.org/developers/x264.html

[17] ITU, "Itu-r bt.601-4 encoding parameters of digital television for studios," The ITU Radiocommunication Assembly, Tech. Rep., 1994.

[18] C. Poynton and G. Johnson, "Color science and color appearance models for cg, hdtv, and d-cinema," in *ACM SIGGRAPH 2004 Course Notes*, ser. SIGGRAPH '04. New York, NY, USA: ACM, 2004. [Online]. Available: http://doi.acm.org/10.1145/1103900.1103903

[19] OpenCV, *The OpenCV Reference Manual*, 2nd ed., OpenCV, December 2012.

[20] (2012) Live555 streaming media. WWW. Live Networks Inc. Last visited: 20130227. [Online]. Available: http://www.live555.com/liveMedia/

[21] (2012) Real time streaming protocol (rtsp). WWW. Internet Engineering Task Force. Last visited: 20130227. [Online]. Available: http://www.ietf.org/rfc/rfc2326.txt

[22] TUIO.org, "Tuio," http://www.tuio.org, 09 2012, accessed September 27, 2012. [Online]. Available: http://www.tuio.org

[23] M. Kaltenbrunner, T. Bovermann, R. Bencina, and E. Costanza, "Tuio: A protocol for table-top tangible user interfaces," in *Proc. of the The 6th International Workshop on Gesture in Human-ComputerInteraction and Simulation*, 2005.