REMOTE RENDERING OF COMPUTER GAMES

Peter Eisert, Philipp Fechteler
Fraunhofer Institute for Telecommunications, Einsteinufer 37, D-10587 Berlin, Germany
eisert@hhi.fraunhofer.de, philipp.fechteler@hhi.fraunhofer.de

Keywords: remote gaming, graphics streaming, 3D coding, networking

Abstract: In this paper, we present two techniques for streaming the output of computer games to an end device for remote gaming in a local area network. We exploit the streaming methods in the European project Games@Large, which aims at creating a networked game platform for home and hotel environments. A local PC based server executes a computer game and streams the graphical and audio output to local devices in the rooms, such that the users can play everywhere in the network. Dependent on the target resolution of the end device, different types of streaming are addressed. For small displays, the graphical output is captured and encoded as a video stream. For high resolution devices, the graphics commands of the game are captured, encoded, and streamed to the client. Since games require significant feedback from the user, special care has to be taken to achieve these constraints for very low delays.

1 INTRODUCTION

Computer games are a dynamic and rapidly growing market. With the enormous technical development of 3D computer graphics performance of nowadays home computers and gaming devices, computer games provide a broad range of different scenarios from highly realistic action games, strategic and educational simulations to multiplayer games or virtual environments like Second Life. Games are no longer a particular domain of kids but are played by people of all ages. Games offer also leisure time activity at home, for guests in hotels, and visitors in Internet Cafes.

Modern games, however, pose high demands on graphics performance and CPU power which is usually only available for high end computers and game consoles. Other devices such as set top boxes or handheld devices usually lack the power of executing a game with high quality graphical output. For ubiquitous gaming in a home environment, a hotel, or a cafe, however, it would be beneficial to run games also on devices of that kind. This would avoid placing a noisy workstation in the living room, or costly computers in each room of a hotel. This problem could be solved by executing the game on a central server and streaming the graphics output to a local end device like a low cost set top box. Besides the ability to play games everywhere in the entire network such a scenario could also benefit from load balancing when running multiple games simultaneously.

In this paper, we present first investigations for streaming games' output over a local area network. We consider two different approaches: video streaming of an already rendered frame of the game and the streaming of graphics commands and local rendering on the end device. Both approaches are part of the European project Games@Large (Tzruya et al., 2006) that develops a system for remote gaming in home and hotel environments. The paper is organized as follows. First, we present the architecture of the gaming system. Then, we investigate the usage of video coding techniques for streaming graphical content and illustrate the differences for game scenarios. In Section 4 we finally depict our system for the streaming of graphics commands, which can be exploited for transmission of high resolution content.

2 ARCHITECTURE OF THE SYSTEM

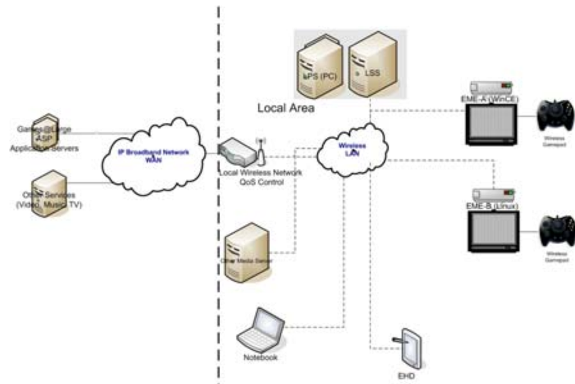

Figure 1: System architecture.

The Games@Large system targets at providing a platform for the remote gaming in home, hotel, and other local environments. The architecture of the system is depicted in Fig. 1. The core of the system is a PC that executes the game. No special game adaptations are necessary, but any commercial game can be played. The user just selects the desired game from a web site. In order to avoid a local installation of the game, it runs in a virtual machine. An image of the game environment is downloaded from a provider. Since important parts of the data are transmitted first, the game can be started before the download is completed.

Games usually require severe constraints on computational power and graphics capabilities. Since such high-end PCs are not available in each room of a household or a hotel, video, audio, and graphics streaming over local networks is used to enable ubiquitous gaming. The output of the game, executed on the server, is grabbed and sent to different end devices. These can be smaller PCs or laptops, but also much cheaper set top boxes or handheld devices. Dependent on the capabilities of the end device, different streaming techniques are used. For devices with small displays like handhelds or PDAs, which usually do not have hardware graphics support, video streaming is applied, whereas devices having a graphics card are supplied directly with the graphics commands and render the graphics of the game locally. The two approaches of streaming game content to the end devices is described in more detail in the following two sections.

3 VIDEO STREAMING OF SYNTHETIC CONTENT

One solution to stream the visual output of the game to the end device is the use of video streaming techniques as shown in Fig. 2. Here, the server renders the computer graphics scene, the framebuffer is captured, eventually downsampled to match the target resolution, and the current image is encoded using a standard video codec (Stegmaier et al., 2002). This solution has the advantage, that also end devices with no hardware graphics capabilities can be supported. Decoding video is usually computationally not very demanding and can be performed even on small devices like PDAs or mobile phones. Also, the bit-rate for streaming the graphics output is rather predictable and not fully influenced by the complexity of the graphics scene. On the other hand, encoding a video leads to high computational load at the server which has to be shared with the execution of the game. Especially if multiple games run in parallel on the server, video encoding might be less applicable and graphics streaming could be the better choice. Therefore, we intent to exploit video streaming technologies only to support devices with small displays, where video encoding is less demanding. Other devices like PCs or set top boxes are connected using graphics streaming as described in Section 4.

3.1 Video Coding Performance

Figure 3: Games used for analyzing the different codecs for streaming synthetic games content.

For the analysis of video coding techniques for the use of streaming game output to end devices, we have performed some experiments using MPEG-4 (MPEG-4, 1999) and H.264 (MPEG-4 AVC, 2003) codecs. The output of different games as shown in Fig. 3 was grabbed and encoded. Fig. 4 depicts the rate-distortion plot for different games using H.264 at 4CIF resolution and 30 fps. It can be seen that bit-rates vary between the scenes dependent on the amount of motion and explosions of the game.

The computational complexity, however, for encoding 4CIF with H.264 is rather demanding. For

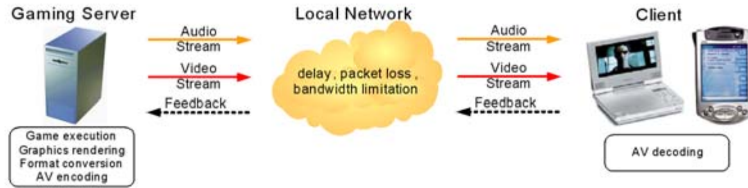

Figure 2: Video streaming from the gaming server.

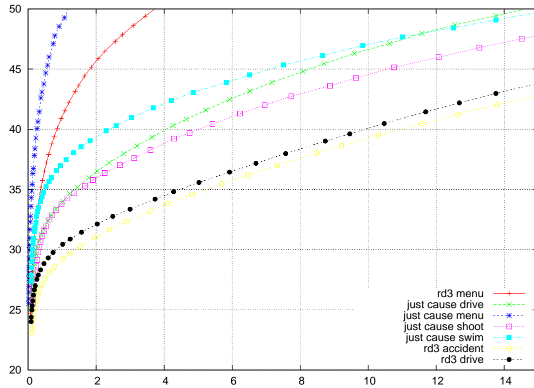

Figure 4: Rate-distortion plot for encoding different game output with H.264 in 4CIF resolution, 30 fps. The curves show a large variability of bit-rates between different scenes.

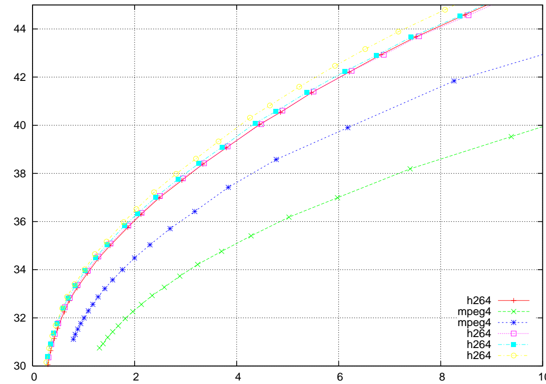

Figure 5: Rate-distortion plot for encoding a game sequence at 4CIF resolution, 30fps with different profiles of MPEG-4 and H.264.

our implementation, frame rates of only 9 fps could be achieved on a 2 GHz Pentium 4. The complexity and performance of different codecs is illustrated in Fig. 5. The upper curves with the best coding performance refer all to different settings of H.264, the two lower ones to MPEG-4. However, the lower quality results also in a lower complexity, and frame rates of 80 fps and 110 fps, respectively, can be achieved for our MPEG-4 implementations.

Streaming games' output, however, is somewhat different than streaming real video. First, the synthetic content is usually free of noise and has different statistics compared to real video. Also the requirement on extremely low delay are much higher in order to enable interactive gaming. On the other hand, additional information about the scene is available from the render context, like camera data or motion and depth information. We exploit that information to reduce the complexity of the H.264 encoding in order to combine high coding efficiency with lower encoding complexity.

4 GRAPHICS STREAMING

The second approach exploited for streaming the game's output is to directly transmit the graphics commands to the end device and render the image there (Buck et al., 2000; Humphreys et al., 2002; Ignasiak et al., 2005). For that purpose, all calls of the OpenGL or DirectX library are intercepted, encoded and streamed. In this framework, encoding is much less demanding and independent from the image resolution. Therefore, high resolution output can be created and parallel game execution on the server is enabled. Also, the graphics card of the server has not to be shared among the games, since rendering is performed at the remote client. On the other hand, bit-rates are less predictable and high peaks of data-rate are expected, especially for scene changes, where a lot of textures and geometries have to be loaded to the graphics card. Also, hardware support for rendering is required at the end device and an adaptation of the game's output to the capabilities of the end device is necessary, which means that not all games can be supported in this mode. In the next section, some statistical analysis on issues related to graphics streaming is presented as well as some description of the current graphics streaming implementation.

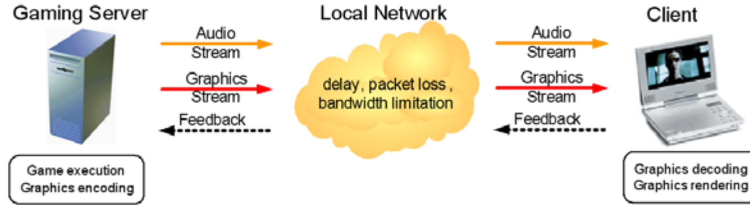

Figure 6: Graphics streaming from the gaming server.

5 STREAMING OF OPENGL GAMES

In this paper, we concentrate on OpenGL games in a Linux environment. Examples of supported games are shown in Fig. 7. Besides direct usage of OpenGL commands, we also support the SDL (Simple Direct-Media Layer) library, which is often used for game programming. All calls to the OpenGL and SDL library are intercepted as well as the *glXSwapBuffers* and *SDL_GL_SwapBuffers* command in order to determine if the frame is ready for display. In order to support the dynamic loading of graphics commands and OpenGL extensions, also the functions *glXGetProcAddressARB* and *SDL_GL_GetProccAddress* are replaced by new versions.

All the graphics commands are streamed to the client and rendered there. Fig. 8 shows the bit-rate (measured in bits per frame) for the raw commands for different game scenarios. It can be seen that there is a high variability of bit-rates. Especially at scene changes, new textures and display lists have to be streamed, leading to extreme peaks of bit-rate. In order to equalize them, intelligent caching and prefetching of textures are required by analyzing the game structure in advance and using side information of the games' provider. Also encoding of textures, geometries (Müller et al., 2006) and graphics commands is added.

5.1 Commands with Feedback

Another issue of streaming graphics, which is rather problematic, are commands that require a feedback from the graphics card. These can be requests for capabilities of the graphics card, but also current state information like the actual projection or modelview matrix. Fig. 9 depicts the temporal change of the number of graphics commands that require some return values from the graphics card. The curves show, that for some frames more that 1000 commands ask for feedback (especially at scene changes), but also during normal game play, several commands requesting feedback are issued. Average and maximum values of bit-rate and commands are also illustrate in Table 1. Since we do not know, what the game does with the return values, we have to wait in our streaming environment until the answer has been returned. This could introduce enormous round trip delays in the client-server structure and makes interactive gaming impossible.

game	gltron	penguin	scorched
avg. bit-rate	51 kbit	261 kbit	227 kbit
max bit-rate	2.1 Mbit	26.6 Mbit	3.6 Mbit
avg. no cmds	3300	14700	13000
max no cmds	37000	41000	106000
avg. no return	4.1	1.5	23.2
max no return	47	1410	1380

Table 1: Command statistics and bit-rate for the three games *gltron*, *penguin racer*, and *scorched3d*. All numbers are measured per frame.

5.2 Local Simulation of OpenGL State

In order to avoid waiting for the client returning results from the graphics card, we simulate the current state of the graphics card at the server and can therefore reply directly without sending the command to the client. We distinguish between different types of commands:

- Static values
- Return values that can be predicted
- Values simulated locally at server.

All static properties, which usually refer to the capabilities of the graphics card, are initially tested at the client and sent to the gaming server. This includes information about maximum texture size, number of display lists, etc. Also, the support of OpenGL extensions can be initially tested once. After the static information has been received at the server, all requests for this data can then be answered locally without sending the command to the client.

Other commands like glGetError can be predicted and dummy return values can be provided. Assuming,

Figure 7: Three different OpenGL games used for the statistical analysis and experiments.

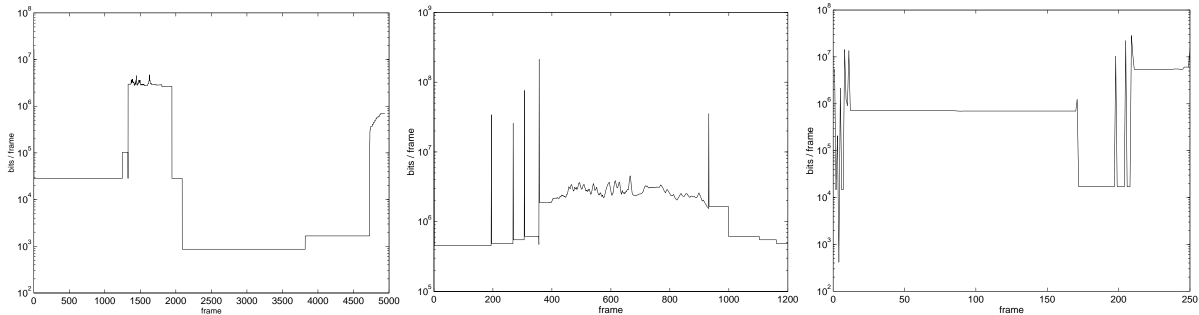

Figure 8: Bit-rate in bits per frame of the uncompressed graphics stream.

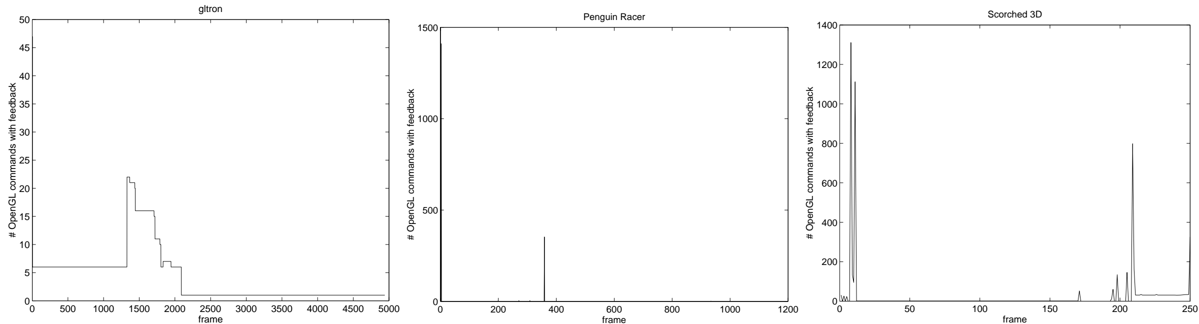

Figure 9: Number of graphics commands that require a feedback from the graphics card.

e.g., that no error has occurred, can avoid the round trip time for such commands. This might not work for all games, but did not lead to problems for the games analyzed here.

However, most of the graphics states change regularly and require a special treatment. In our implementation, we simulate this state in software in parallel to the graphics card. All commands that affect and change the state in the graphics card, initiate the corresponding changes also at the state simulated at the server. One example is the request for the current ModelView or Projection matrix. These matrices are updated locally each time a command like *glRotate*, *glTranslate*, *glScale*, *glLoadMatrix*, *glPushMatrix*, etc. is called. Since these operations do not occur that often, the overhead for doing that in software is rather moderate. However, the round trip delay caused by the game asking for the current matrix can be avoided. Similarly, the other graphics states are simulated such that no single command requiring feedback is executed during normal play of the analyzed games.

5.3 Streaming of Graphics Commands

All graphics commands which need to be send to the client, are encoded, packetized and streamed to the end device. Encoding is currently rather simple and byte oriented. Colors, texture coordinates, and normals can for example be represented by short codewords and need not be represented by multiple floats or doubles. Also textures are encoded using some

transform coding based on the H.264 interger transform. Currently, more efficient coding for textures, geometries, and graphics commands are under investigation. The command itself is represented by a token of 1 or 2 bytes length. However, some groups of commands like, e.g., glTexCoord, glNormal, glVertex that occur often jointly are represented by a single token in order to reduce the overhead.

The commands are then packetized for transmission. For that purpose, it is assured that a command with all its arguments will reside in the same packet, except for commands that contain pointers to large memory areas. Once a packet is full, it is sent to the client. Currently, we use a reliable TCP connection which is not optimal for minimizing delay. However, in contrast to the video streaming approach, that can start with encoding of the video first if the frame buffer has been rendered completely, we can start with the transmission of packets bevor the current frame is processed completely minimizing the delay by almost one frame.

5.4 Experimental Results

The proposed streaming system has been tested with different OpenGL games like *scorched3d*, *gltron*, *penguinracer*, *OpenArena*, *neverball* and the *FlightGear* simulator. When streaming all graphics commands directly without any compression and graphics state simulation, the games are not interactively playable, since delay is much too high. With our proposed system, for locally simulating the graphics card's state, the number of commands during normal game play requiring a feedback could be reduced to zero. This leads to a significant reduction in delay and enables the ability to play the tested games in a local area network. Graphics command compression was enabled but in the current version rather moderate. For the arguments of the commands, an average compression of about 30 % was achieved. The size of the target window can, however, be varied interactively (by manipulating the glViewport command) and need not be the same as the game's resolution. For the considered games, interactive frame rates of up to 30 fps for the simpler games was achieved. For more demanding games like OpenArena, delays are relatively visible. Here, we work on reducing the bitrate by more efficient compression methods and thus reducing the delay for enhanced gaming.

6 CONCLUSIONS

We have presented in our paper a system for the remote gaming in local networks. The architecture of the proposed system is targeted for an execution of commercial games in a virtual environment and ubiquitous gaming due to different streaming techniques. Both, video streaming and transmission of graphics commands are investigated. First analysis shows the applicability of the approaches for different end devices. In order to reduce delay for the graphics streaming, a simple real-time compression of graphics commands and a local simulation of the graphics state has been implemented. This resulted in a significant reduction of delay which is a prerequisite for interactive gaming.

REFERENCES

Buck, I., Humphreys, G., and Hanrahan, P. (2000). Tracking graphics state for networked rendering. In *Proc. SIGGRAPH/EUROGRAPHICS Workshop On Graphics Hardware*.

Humphreys, G., Houston, M., Ng, R., Frank, R., Ahern, S., Kirchner, P. D., and Klosowski, J. T. (2002). Chromium: a stream-processing framework for interactive rendering on clusters. In *Proc. International Conference on Computer Graphics and Interactive Techniques*.

Ignasiak, K., Morgos, M., and Skarbek, W. (2005). Synthetic-natural camera for distributed immersive environments. In *Proc. Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS)*, Montreux, Switzerland.

MPEG-4 (1999). *ISO/IEC 14496-2: Coding of audio-visual objects - Part 2: Visual, (MPEG-4 visual)*.

MPEG-4 AVC (2003). *Advanced video coding for generic audiovisual services, ITU-T Rec. H.264 and ISO/IEC 14496-10 AVC*.

Müller, K., Smolic, A., Kautzner, M., Eisert, P., and Wiegand, T. (2006). Rate-distortion-optimized predictive compression of dynamic 3D mesh sequences. *Signal Processing: Image Communication*, 21(9):818–828.

Stegmaier, S., Magallón, M., and Ertl, T. (2002). A generic solution for hardware-accelerated remote visualization. In *Proceedings of the Symposium on Data Visualisation 2002*.

Tzruya, Y., Shani, A., Bellotti, F., and Jurgelionis, A. (2006). Games@large - a new platform for ubiquitous gaming and multimedia. In *Proceedings of BBEurope*, Geneva, Switzerland.