LOW DELAY STREAMING OF COMPUTER GRAPHICS

P. Eisert and P. Fechteler

Fraunhofer Institute for Telecommunications Heinrich-Hertz Institute
Einsteinufer 37, 10587 Berlin, Germany
email: {eisert,philipp.fechteler}@hhi.fraunhofer.de

ABSTRACT

In this paper, we present a graphics streaming system for remote gaming in a local area network. The framework 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 output to local devices in the rooms, such that the users can play everywhere in the network. Since delay is extremely crucial in interactive gaming, efficient encoding and caching of the commands is necessary. In our system we also address the round trip time problem of commands requiring feedback from the graphics board by simulating the graphics state at the server. This results in a system that enables interactive game play over the network.

Index Terms— graphics streaming, remote gaming, 3D coding

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 a low delay system for streaming graphics over a local area network with the special application of computer games. One approach often used to display graphics output remotely, is to render the graphics locally, grab the framebuffer content, encode the frame using a standard video codec like H.264 [1] and to transmit the output to the client as a video stream [2, 3, 4]. This has the advantage of predictable bit-rates, no requirement of hardware accelerated rendering at the client, and independence of the graphics scene complexity and features. However, video encoding of high-resolution output is computationally very demanding and has to run in parallel on the same PC with the computer game. With that technique, the execution of multiple games on one server is extremely difficult since this would require parallel encoding and the games would have to share a graphics card. Also, delay, a crucial issue for action games, might be higher since streaming of the video can first be started after having rendered and grabbed the entire frame.

An alternative approach is to directly transmit the 3-D graphics and to render at client side. There exist already several standards for graphics compression like, e.g. X3D [5] or the 3D Graphics Compression Model [6] of MPEG-4 which includes Frame-based Animated Mesh Compression [7] as well as Bone-Based Animation and scene graph compression. However, in this paper, we address the streaming of graphics for already existing applications (e.g. games) where the way how graphics is represented cannot be influenced and might change completely from frame to frame. Therefore, we directly intercept the graphics commands being sent to the graphics library like OpenGL or Direct3D. These commands are encoded and streamed to the client device. Although this requires a graphics chip at the end device, we avoid the encoding complexity of video compression at the server as well as the usage of its graphics board. Therefore, multiple games can be executed on one PC necessary for providing game services at hotels or other public places. Also, we can start streaming graphics content as soon as the first commands arrive and do not have to wait for the entire frame to be rendered. On the other hand, high bit-rates may arise for the transmission of graphics commands, objects and textures, which have to be compressed in order to enable real time streaming as presented in this paper.

2. ARCHITECTURE OF THE SYSTEM

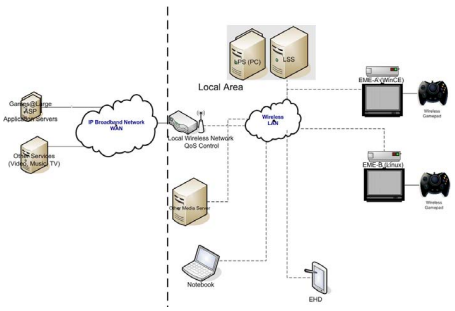

Fig. 1. System architecture.

The graphics streaming system is embedded into a larger framework for remote gaming that is developed in the European project Games@Large [8]. This 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. Although this paper deals only with the graphics streaming part, the framework also provides a fallback solution by means of video streaming for small end-devices like handhelds or PDAs, which usually do not have hardware graphics rendering capabilities.

3. GRAPHICS STREAMING

Fig. 2. Graphics streaming from the gaming server.

The approach exploited in this paper for streaming the game's output is to directly transmit the graphics commands to the end device and render the image there [9, 10, 11]. For that purpose, all calls of the OpenGL or DirectX library are intercepted, encoded and streamed. In this framework shown in Fig. 2, 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. 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. 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.

4. STREAMING OF OPENGL GAMES

In this section, we will show exemplarily for some simple games like the ones depicted in Fig. 3 what kind of problems may arise for the streaming of graphics output. We will concentrate on OpenGL games in a Linux environment, for Windows and DirectX similar results apply. We use our implementation of the graphics streaming system to protocol some statistics on the number of commands and bit-rate.

For that purpose, the system intercepts all calls to the OpenGL library as well as the SDL (Simple Direct Media Layer) library, which is often used for game programming. Also the glx command *glXSwapBuffers* and the corresponding *SDL_GL_SwapBuffers* are modified and point to the new library 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.

The games are played and all the graphics commands are streamed to the client where they are rendered for display. In parallel, the number of commands are recorded and bit-rates are measured for each frame. Fig. 4 shows the bit-rate (measured in bits per frame) for the raw commands for the game *penguinracer*. 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

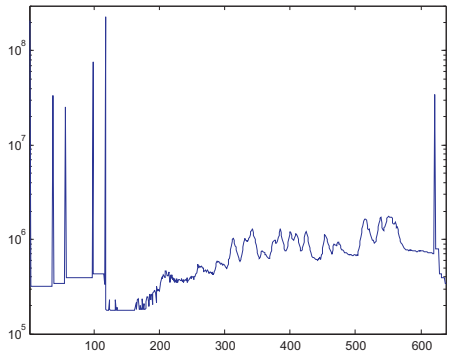

Fig. 4. Bit-rate in bits/frame for the game *penguinracer*. High peaks in bit-rate correspond to scene and level changes.

to extreme peaks of bit-rate. If these peaks are at level changes they might not degrade the game play too much but even during game play higher bit-rates arise that need efficient encoded to enable interactive streaming over existing networks.

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. Measurements for different games 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. Since we do not know, what the game does with the return values, we have to wait in a naive streaming environment until the answer has been returned. This would introduce enormous round trip delays in the client-server structure and makes interactive gaming impossible.

5. GRAPHICS ENCODING AND CACHING

In order to achieve low delay streaming of OpenGL graphics commands in game applications, we address the following issues

- caching of server memory at the client
- local simulation of graphics state
- encoding / compression of graphics stream

which are described in the following subsections.

5.1. Caching of Objects at the Client

Some commands like the vertex arrays only send a pointer to main memory to the graphics board, which then fetches data for object geometry very efficiently. However, in a streaming environment, the main memory resides at the server hosting the game, which the graphics board at the client cannot access. Therefore, the memory of the buffers need to be transmitted to the client, where they are cached for future use. Since the game can change the memory without notifying the graphics hardware, these changes have to be detected and the client's memory need to be updated accordingly. This is achieved by buffering the arrays at the server as well, such that the memory can be compared with the current state of the buffers. Before a command like e.g. *glDrawElements* is executed, memory updates might need to be added to the command string. Locking an array avoids the search for memory modifications and the update of the client space increasing server efficiency. With this technique, vertex, normal, texture, etc. arrays are cached, but also index lists used e.g. by *glDrawElements* are stored in the client for future use. In order to keep the memory requirements for the caches moderate,

Fig. 3. Three different OpenGL games used for the statistical analysis and experiments.

the access frequencies of the memory areas are measured and used to delete rarely used arrays.

5.2. Local Simulation of OpenGL State

The most important issue for obtaining a low delay streaming system is the avoidance of any commands the require some feedback from the graphics card. In order to avoid waiting for the client returning results, 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, 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 any of 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.

Another advantage of simulating the graphics state is that many commands need not be sent at all. Usually, the application sets all states required for rendering the next object independent on their previous state. Therefore, many commands do not actually change the state since it has been set appropriately by some previous commands. Since we know the current state of graphics board, we can check whether a command change some properties or not. In the latter case, the command is dropped and not sent over the network.

One particular issue, however, complicates the state tracking, which is the concept of display lists. Display lists are basically macros of graphics commands which can be executed simply be referring to the list identifier. Since they are executed on the client, they can change the graphics state without notifying the server by individual commands. Therefore, we also keep track of state changing commands during the creation of display lists. Since attributes can be saved by using push and pop commands and only changes after the execution of display lists need to be monitored, only a few commands need to be tracked. Once a display list is executed, the corresponding reduced command stream is also executed at the server to keep the state consistent with the client.

5.3. Encoding of Graphics Stream

The caching described in the previous subsection avoids a large amount of data from being sent over the network. The remaining data consists of graphics commands that influence the state machine and additional object data like, e.g., textures. Textures are separately coded with a 4 by 4 integer transform like in H.264. The graphics command stream is simply encoded with a hybrid coding structure similar to video coding with intra I and predictive P frames. This can be done, since many objects are rendered each frame in a similar way and the command string shows a repetitive pattern. The first frame is encoded independently whereas the following frames exploit information from the previous one. Since we target for real-time encoding of large amount of data, the encoding is rather simple and mostly byte oriented. Future work will concentrate on more elaborate coding methods.

In a first step, the arguments are quantized dependent on their content. Colors, texture coordinates, and normals can for example be represented by short codewords and need not be represented by multiple floats or doubles. 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. Texture and image data form a second stream and are handled differently.

The slightly compressed graphics stream (without texture and image data) is stored at the server for reference when encoding the next frame. When the next P frame is encoded, blocks of graphics commands that have already occurred in the previous frame are encoded simply by referencing them by their start index and length similar to motion vectors in video coding. Many objects and command strings can thus be handled efficiently without retransmission. The only problem is the efficient search for similar patterns with real-time constraints.

Here, we exploit the fact that the scenes are often rendered in a tree-like manner and that the commands have a particular meaning.

Thus we restrict our search by looking for correspondences of a very small set of commands like *glPushMatrix*, *glPushAttrib*, *glBegin*, *glMaterial* that might start the definition of new objects or a vertical movement in the scene graph. Starting from these commands (in a limited search range) similar byte patterns are searched and signaled in the new stream.

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 before the current frame is processed completely minimizing the delay by almost one frame.

5.4. Experimental Results

game	penguin	OpenArena	FlightGear
avg. bit-rate (Mbit/frame)	2.7	4.0	5.2
max bit-rate (Mbit/frame)	229	42	847
avg. # cmds	7718	1654	11600
max # cmds	21600	3711	25400
avg. # return	2.6	0.4	21
max # return	1324	73	1480
bit-rate saving cache	66%	58%	73%
bit-rate saving total	84%	68%	85%

Table 1. Bit-rate and number of graphics commands (total and only commands with request for return values) for the three games *penguin racer*, *OpenArena*, and *Flight Gear*. All numbers are measured per frame.

The proposed streaming system has been tested with different OpenGL games like *penguinracer*, *OpenArena*, a quake3 based first person shooter game, and the *FlightGear* simulator. OpenGL commands are currently implemented up to version 1.3 as well as some extensions. 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. This can be seen from Table 1, where measured average and maximum bit-rate (in bits per frame) of the uncoded graphics stream are depicted together with the number of graphics commands issued per frame. Some frames show more than 1000 requests (mostly at level changes) to the graphics board but some games require feedback even during normal game play. With our proposed system for locally simulating the graphics card's state, the number of commands during normal game play requiring a feedback has been 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 not yet optimized. Most reduction of bit-rate has been achieved by the caching of geometry data at the client's memory. Table 1 show bit-rate savings between 60 and 80 % during normal game play. These bit-rate savings do not include the initialization phase of the game with loading all the textures. The remaining graphics command stream is then encoded using information from the previous frame as described in Section 5.3. Some frames can be compressed to less than 5% of the remaining bit-rate but in average only a moderate bit-rate reduction of about 30% was achieved. Combining both techniques, bit-rate savings of 70 – 85% have been measured. The

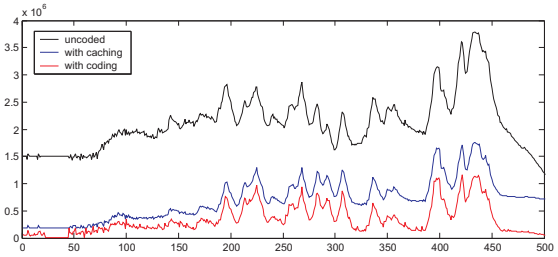

Fig. 5. Bit-rate in bits/frame for the game *penguinracer* during game play. The upper curve denotes the uncoded bit-rate, while the lower two show the results for activated caching and additional encoding, respectively.

bit-rate reduction is also shown exemplarily in Fig. 5 for the game *penguin racer*. Here, only normal game play is considered without the initialization and startup menu phase depicted in Fig. 4. It can be seen, that bit-rate is reduced significantly for all frames when activating the presented caching and coding methods.

6. CONCLUSIONS

We have presented a system for remote gaming in local area 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. First analysis shows the applicability of the approach for different games. 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. With the proposed system, bit-rate saving of about 80 % and a significant reduction of delay is achieved which is a prerequisite for interactive gaming.

7. REFERENCES

[1] *Advanced video coding for generic audiovisual services, ITU-T Rec. H.264 and ISO/IEC 14496-10 AVC*, 2003.

[2] S. Stegmaier, M. Magallón, and T. Ertl, "A generic solution for hardware-accelerated remote visualization," in *Proceedings of the Symposium on Data Visualisation 2002*, 2002.

[3] D. De Winter, P. Simoens, L. Deboosere, F. De Turck, J. Moreau, B. Dhoedt, and P. Demeester, "A hybrid thin-client protocol for multimedia streaming and interactive gaming applications," in *Proc. ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video NOSSDAV'06*, Rhode Island, USA, May 2006.

[4] P. Eisert and P. Fechteler, "Remote rendering of computer games," in *Proc. Internation Conference on Signal Processing and Multimedia Applications (SIGMAP)*, Barcelona, Spain, Jul. 2007.

[5] *ISO/IEC 19775:2004: Information technology Computer graphics and image processing Extensible 3D (X3D)*, 2004.

[6] *ISO/IEC 14496-25: MPEG-4 Part 25: 3D Graphics Compression Model*, 2008.

[7] *ISO/IEC 14496-16: MPEG-4 Part 16 AMD2: Frame-based Animated Mesh Compression*, 2007.

[8] Y. Tzruya, A. Shani, F. Bellotti, and A. Jurgelionis, "Games@large - a new platform for ubiquitous gaming and multimedia," in *Proceedings of BBEurope*, Geneva, Switzerland, Dec. 2006.

[9] I. Buck, G. Humphreys, and P. Hanrahan, "Tracking graphics state for networked rendering," in *Proc. SIGGRAPH/EUROGRAPHICS Workshop On Graphics Hardware*, 2000.

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

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