ACCELERATED VIDEO ENCODING USING RENDER CONTEXT INFORMATION

Philipp Fechteler[1] *and Peter Eisert*[1,2]
{philipp.fechteler, peter.eisert}@hhi.fraunhofer.de

[1]Fraunhofer Heinrich Hertz Institute
Image Processing Department
Einsteinufer 37, D-10587 Berlin, Germany

[2]Humboldt Universität zu Berlin
Department of Computer Science
Unter den Linden 6, 10099 Berlin, Germany

ABSTRACT

In this paper, we present a method to speed up video encoding of GPU rendered 3D scenes, which is particularly suited for the efficient and low-delay encoding of 3D game output as a video stream. The main idea of our approach is to calculate motion vectors directly from the 3D scene information used during rendering of the scene. This allows the omission of the computationally expensive motion estimation search algorithms found in most of today's video encoders. The presented method intercepts the graphics commands during runtime of 3D computer games to capture the required projection information without requiring any modification of the game executable. We demonstrate that this approach is applicable to games based on Linux/OpenGL as well as Windows/DirectX. In experimental results we show an acceleration of video encoding performance of approximately 25% with almost no degradation in image quality.

Index Terms— 3D Computer Graphics, Video Encoding, Video Streaming, Game Streaming, Games on Demand

1. INTRODUCTION

Remote visualization of interactive 3D applications with high quality and low delay is a long-standing goal. The advent of commercially available games-on-demand systems, like OnLive, Gaikai, t5 labs etc. emphasizes the importance of efficient streaming methods for interactive 3D computer games. Generally, there are two approaches frequently used to stream 3D graphics content: a) transmission of the graphics commands, objects and textures, and rendering of the images on client side, or b) rendering the images on the server side, and sending them as encoded video sequence. This latter method is currently used by all commercial gaming-on-demand systems. Several systems are available for both approaches, but most of them introduce a delay, which makes them unsuitable for highly interactive applications like computer games.

In this paper we also focus on the second approach, i.e. low-delay video streaming of interactive 3D content with the application rendering its output on the server side. The synthesized images are grabbed from the framebuffer and encoded as video, using the current state of the art video codec H.264/AVC. In video coding, temporal correlation is mostly exploited by block-based motion compensation, where the macroblocks of the current frame are predicted from previously encoded frames. The removal of such correlations yields high compression ratios, but the search for the best motion vectors is a computational expensive process. Clever search methods have been developed in the past, but motion vector estimation still contributes a major expense in video encoding. This paper deals with the optimization of motion estimation for the particular case of synthesized graphics by exploiting additional information from the render context.

Several approaches for remote access to interactive 3D computer graphics applications are reported in the literature. In [1], the authors present a streaming protocol which streams video data for GPU based and high-motion applications and low level graphics commands when little motion happens. In [2], a method is presented where foreground and background objects are encoded with different quantizations by taking graphics information into account.

2. APPLICATION FRAMEWORK

The proposed method is embedded into a larger framework for remote gaming that is developed in the European project Games@Large [3]. This system targets at providing a platform for remote gaming in homes, hotels, internet cafes, and other local environments. Instead of executing the game locally on a PC, e.g. in the hotel room, it runs on a central server and the graphical and audio output is distributed over a LAN to smaller and cheaper end devices, like settop boxes, notebooks, or hand held devices. Since commercial games shall be supported off-the-shelf, no modifications to the game code can be accepted.

The overall system supports two modes for the transmission of the games graphical output. For high resolution content, graphics streaming is used. For that purpose, the graphics commands are intercepted before they reach the graphics library OpenGL or DirectX. The commands are encoded, streamed to the client and rendered there [4]. The second mode available is video streaming, and it is used to transmit graphical output of lower resolutions to smaller end-devices, like handhelds or PDAs, which usually do not have hardware graphics rendering capabilities. The challenges for this approach are to achieve a very low delay, in order to handle interactive gaming over networks, and the reduction of encoding complexity, since the encoding has to be executed in parallel to the running game. This paper describes one method for this complexity reduction employed within the framework.

3. ACCELERATED MOTION VECTOR ESTIMATION

In [5], we have presented a method to capture skybox information of games to directly calculate the corresponding motion vectors, see the blue regions in Fig. 1 and 2. The method presented here complements this approach by capturing the projection information of the most dominant objects in a scene (middle image of Fig. 1).

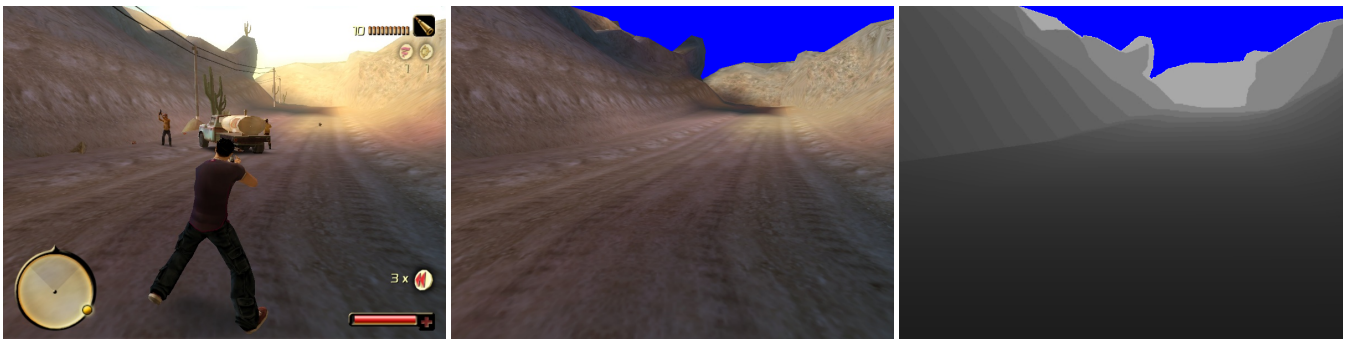

Fig. 1. **left:** original game output **middle:** scene corresponding to dominant motion **right:** corresponding depth map - detected skybox regions are in blue ink

3.1. Efficient and Direct Calculation of Motion Vectors

In order to calculate directly the motion vectors for a given macroblock (partition), the corresponding 3D location is computed by back-projecting the center location of the macroblocks, using screen coordinates and the depth buffer value together with the current projection matrices used by the game to render this scene. The resulting 3D location is then re-projected to the pixel location of the previous frame using previous projection matrices. The motion vector is the difference of both locations, scaled by four, to account for quarter-pixel resolution of H.264/AVC.

In contrast to [6], we do not use `gluUnProject()` and `gluProject()` of the OpenGL Utility Library, nor their counterparts in DirectX, i.e. `D3DXVec3Unproject()` and `D3DXVec3Project()`. Instead, we have implemented the motion vector calculation by assembling all pixel independent calculations, like normalization, matrix-inversion and matrix-matrix-multiplication into a single matrix being constant for the entire frame. As a result, the calculation of the previous location corresponding to the current macroblock center is just a single multiplication of a 3×4-matrix with a 4-vector:

$$\begin{aligned} p_{prev} &= N^{-1} P_{prev} P_{cur}^{-1} N p_{cur} = M p_{cur} \\ MV &= 4 \cdot (p_{prev} - p_{cur}) = 4 \cdot (M-1) p_{cur} = \tilde{M} p_{cur} \end{aligned} \quad (1)$$

with N being the matrix for normalizing screen coordinates to [-1,1] space and P being the matrix containing all projection information (model view, projection, world transform). Both matrices P and N are specific to DirectX and OpenGL respectively, which is taken into account during calculation of $\tilde{M}$. All these pixel independent computations are combined into one single matrix $\tilde{M}$, which also compensates for the OpenGL and DirectX specifics. Care has to be taken for motion vectors pointing out of the frame or across skybox and scene regions, e.g. by using generic H.264/AVC motion vector search algorithms in such cases. Encountering uncovered regions or object boundary crossings is another problematic case. This could be detected by comparing the calculated depth value for the previous frame and the measured one. Our experiments have shown that handling these effects explicitly does not improve the over-all performance, since the rate-distortion based mode decision, as discussed below, already overcomes this problem.

3.2. Capturing Scene Projection Information

For the direct motion vector calculation, the matrix $\tilde{M}$ is needed. For the region of main scene elements which are most dominant in the current frame, the projection information P_{cur} can be captured efficiently. An example of such a region are mountains in the background, as e.g. seen in the left part of Fig. 1 and 2. This projection information P_{cur} is captured by intercepting all the drawing commands of the running game through Proxy-DLLs, or their OpenGL equivalents [7] respectively. Since each drawing command contains a `count`-parameter to specify the number of vertices to be rendered, it is simple to determine the projection information P_{cur} that is used for the largest object in the current scene, see middle of Fig. 1. In order to calculate $\tilde{M}$, the corresponding matrix P_{prev} of the previous frame has to be determined. Several experiments with different games have shown that a single matrix is always massively dominant in terms of vertices it was used for during rendering, so that this frequency is used to set up the correspondence relation. In more fragile cases, additional hints could be used, like the specific kind of drawing commands, vertex buffer pointers, texture id's etc. In the same way, two or more matrix correspondences can be exploited to provide more than one motion vector candidate for scenes with more sophisticated motion structures.

Capturing the depth map is achieved with a simple `glReadPixels(...GL_DEPTH_COMPONENT...)` invocation for OpenGL based games, see Fig. 3. In case of DirectX, most often the depth buffer is not lockable and therefore not readable. By intercepting the pertinent commands and setting the z-buffer lockable (e.g. `D3DFMT_D32F_LOCKABLE`) the depth map can be read. Our experiments have revealed though, that this modification generates a significant slowdown in game execution and some games even crash or display artefacts. A method to solve this problem is to register all relevant drawing commands (which modify z-values and are not alpha-blended) in a list, and render the depth component of the scene a 2^{nd} time into a dedicated render target. Therefore, during the `EndScene()` command, a simple shader program is loaded to write merely the z-values while rendering the list of collected commands into the separate render target, see Fig. 3. This approach solves an additional problem: some games, like Total Overdoze (see Fig. 1), render into off-screen render targets and finally copy merely the image without its depth component as texture into the screen buffer. In this not unusual case, the depth buffer will be empty.

In Fig. 2 we show an example of predicting the current frame for each pixel separately based on the previous frame, corresponding z-buffers, as well as the projection information for the skybox and the scene. It is clearly visible, that the prediction is accurate for the majority of pixels. Handling of wrong predictions is discussed in the following subsection.

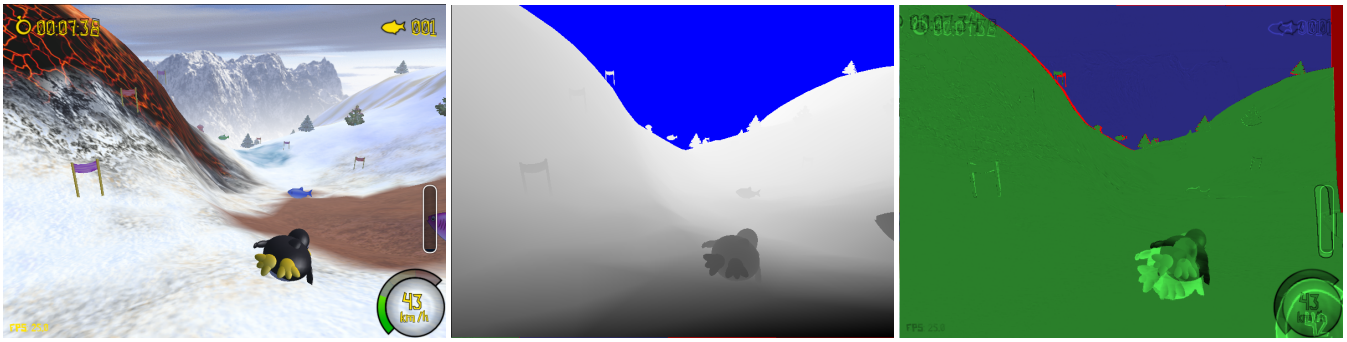

Fig. 2. **left:** original game output **middle:** depth map with blue colored skybox region **right:** difference image between original and prediction with green belonging to scene, blue belonging to skybox and red colored regions are not predictable

3.3. Rate-Distortion based Mode Decision

To actually encode a frame, the frame buffer, as well as the z-buffer and the current projection information assembled into one single pixel- and library-independent matrix $\tilde{M}$ is captured. For each macroblock (partition), it is checked, if it lies entirely in a skybox or non-skybox region. In cases where macroblock partitions contain skybox as well as non-skybox regions, the generic H.264/AVC motion search is utilized. Otherwise, the motion vector is calculated directly according to equation (1). In order to prevent wrongly predicted motion vectors, the rate-distortion costs are considered. In cases where this back- and forth projection is not accurate, e.g. on 2D projections (like speedometer) or exposures, the prediction will be poor, hence the rate-distortion costs will be high. By performing the generic H.264/AVC motion search in cases where the rate-distortion costs exceed a certain threshold, intolerable image quality degradation is prevented. In our experiments, we achieved satisfactorily results by simply setting the threshold to an empirically detected but quantizer-dependent value.

3.4. Further Speedup by GPU Shader Programming

Since the smallest H.264/AVC partitions are of size 4x4, a depth map reduced by a factor of 16 will be sufficient to calculate all desired motion vectors. The scaling could be accomplished efficiently using the MipMap capabilities of the GPU. As a result, the data which needs to be transferred form GPU to CPU is reduced by a factor of 16.

Alternatively, a significant acceleration can be achieved additionally by performing the motion vector calculations directly on the GPU. By exploiting the massive parallelism and the strong floating-point arithmetic capabilities of GPU shader programs, a speedup of more then 10 times is achieved. Additionally, the amount of data that has to be transmitted from GPU to CPU is significantly reduced in this case, since no depth map is needed, but only one motion vector per H.264/AVC macroblock partition. The delay caused by data transmission is therefore further reduced.

4. EXPERIMENTAL RESULTS

In our practical experiments we implemented the scene prediction inside H.264/AVC encoding and compared the encoding time as well as PSNR image degradation with generic H.264/AVC encoding. Computationally demanding settings, such as B-frames, CABAC or multi-picture prediction, were disabled for our tests, since they significantly increase the delay/complexity and are independent of the proposed method. Our test sequences consist of 200 frames in SVGA (800x600) and CIF (352x288) resolution, captured at 25 frames per second. We tested several DirectX and OpenGL games, particularly Total Overdose as DirectX representative and Extreme Tux Racer as representative for OpenGL games.

In Fig. 3 typical timing profiles are depicted for loading the SVGA depth map from GPU to CPU, a) for OpenGL with 16 bit integer depth map format and b) for DirectX with 32 bit float depth map format. Capturing the depth map with OpenGL takes on average 1.1 msec. Due to measurement noise, the time for DirectX depth map capturing appears to vary much more than OpenGL. The rendering of the depth component takes on average 0.1 msec while the loading from GPU to CPU takes 3 msec, which broadly conforms to the OpenGL measurements with half the data volume.

In Fig. 4, we show the results for Extreme Tux Racer at SVGA as well as CIF resolution. At CIF resolution and 35 dB PSNR image quality there is an accelerated encoding performance of 5.8 msec per frame, in comparison to 7.7 msec, showing that the optimized encoding needs approximately only 75% of computation time.

Further accelerations have been achieved using GPU programming techniques. Downscaling the depth map by a factor of 4 with one pixel for each 4x4 H.264/AVC macroblock partition, reduces the transfer time from 1.1 msec to 0.12 msec. With the alternative GPU based approach, calculating all motion vectors directly on GPU, the computation is accelerated by a factor of 10.

5. CONCLUSION

In this paper, we have presented a method to accelerate video encoding for dynamic 3D computer graphics by capturing additional projection information for major scene regions of the visual output. Using this for efficient direct motion vector calculation, an accelerated video encoding is achieved while the image quality degradation is negligible. The speedup is further increased by exploiting GPU programming techniques to accelerate the computations of motion vectors as well as reducing the amount of data needed to be transferred from GPU to CPU. In experiments we demonstrated an overall acceleration of approximately 25%.

6. REFERENCES

[1] D. de Winter et al., "A Hybrid Thin-Client Protocol for Multimedia Streaming and Interactive Gaming Applications," in *Proc. of ACM Intern. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV06)*, Rhode Island, USA, May 2006.

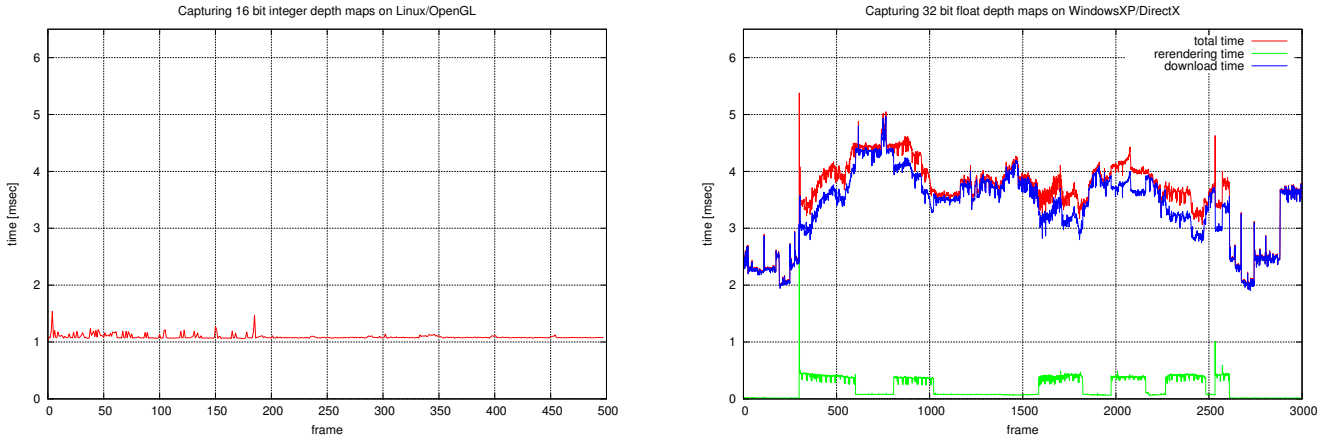

Fig. 3. Timing profiles for capturing SVGA depth buffers, **left:** OpenGL (16 bit integer), **right:** DirectX (32 bit float).

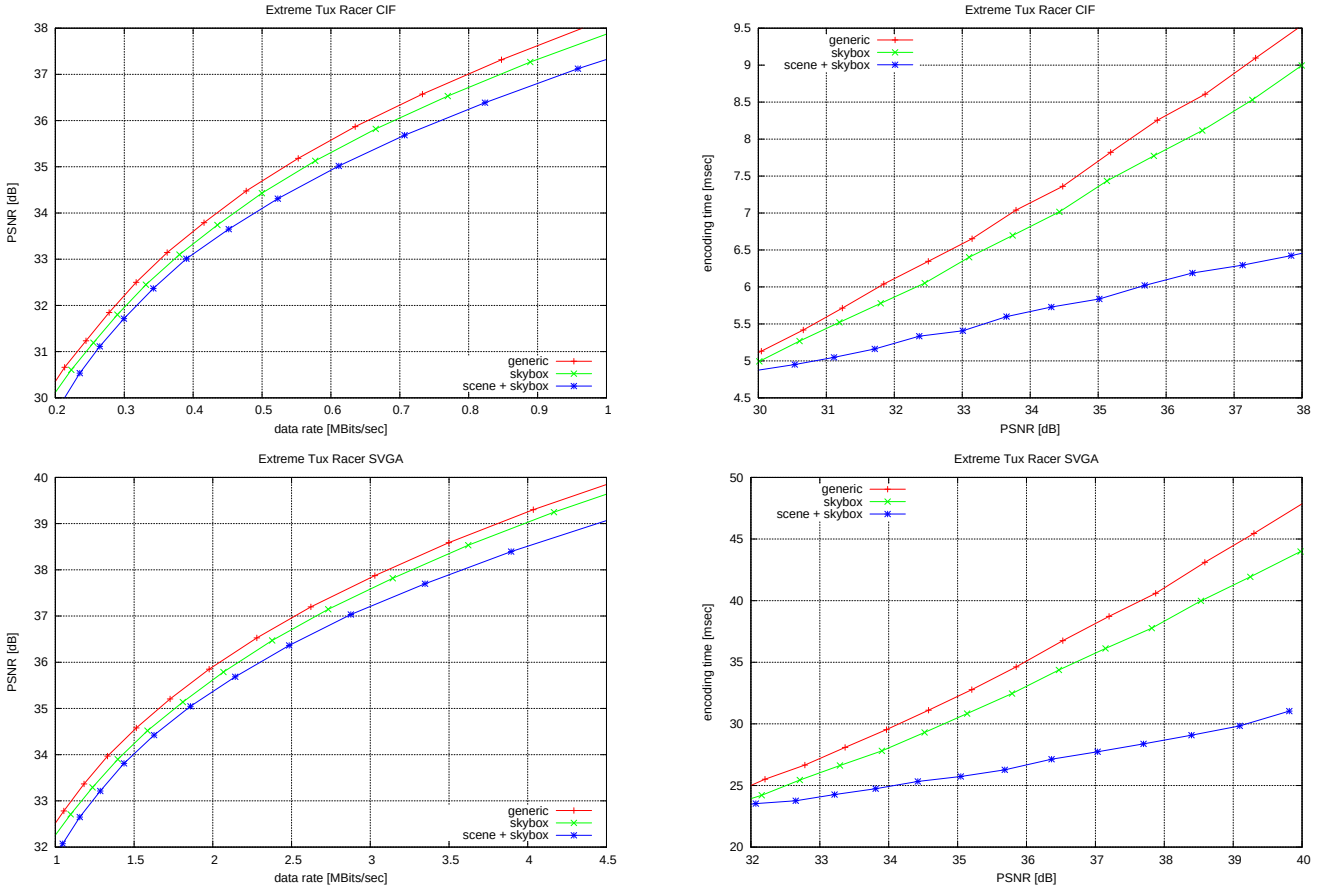

Fig. 4. Influence of PSNR and encoding time.

[2] Y. Noimark and D. Cohen-Or, "Streaming Scenes to MPEG-4 Video-Enabled Devices," *IEEE Computer Graphics and Applications (CGA)*, vol. 23:1, pp. 58– 64, Jan/Feb 2003.

[3] A. Jurgelionis et al., "Platform for Distributed 3D Gaming," *Intern. Journal of Computer Games Technology*, vol. 2009, pp. 15, June 2009.

[4] P. Eisert and P. Fechteler, "Low Delay Streaming of Computer Graphics," in *Proc. of Intern. Conf. on Image Processing (ICIP2008)*, 12-15th Oct. 2008.

[5] P. Fechteler and P. Eisert, "Depth Map Enhanced Macroblock Partitioning for H.264 Video Coding of Computer Graphics Content," in *Proc. of Intern. Conf. on Image Processing (ICIP2009)*, Cairo, Egypt, 7-10th Nov. 2009, pp. 3441–3444.

[6] L. Cheng et al., "Real-Time 3D Graphics Streaming Using MPEG-4," in *Proc. of IEEE/ACM Intern. Workshop on Broadband Wireless Services and Applications (BroadWISE 2004)*, San Jose, CA, USA, 25th Oct. 2004.

[7] S. Stegmaier et al., "Widening the Remote Visualization Bottleneck," in *Proc of Intern. Symposium on Image and Signal Processing and Analysis (ISPA 2003)*, Rome, Italy, 18-20 Sept. 2003, vol. Vol.1, pp. 174– 179.