The Design and Implementation of a VR-Architecture for Smooth Motion

F. A. Smit
Centrum Wiskunde en Informatica
Amsterdam

R. van Liere
Centrum Wiskunde en Informatica
Amsterdam

B. Fröhlich
Bauhaus-Universität
Weimar

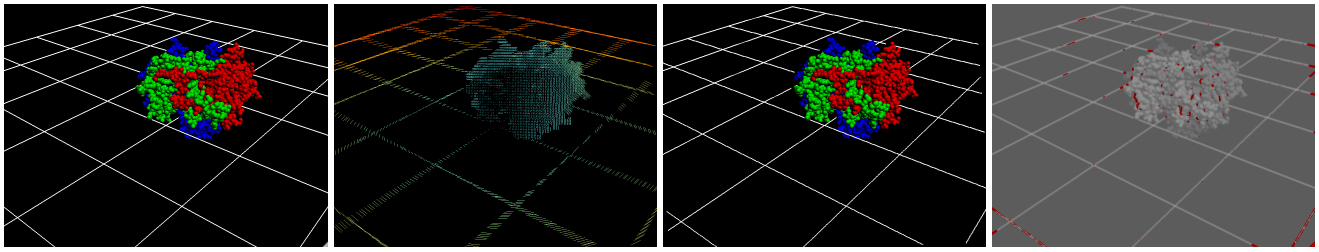

Figure 1: *From left to right: an application frame consisting of a 500k polygon model of a molecular surface, the associated motion field, an extrapolated video frame, and the error due to extrapolation. The camera is rotating around the molecule clockwise.*

Abstract

We introduce an architecture for smooth motion in virtual environments. The system performs forward depth image warping to produce images at video refresh rates. In addition to color and depth, our 3D warping approach records per-pixel motion information during rendering of the three-dimensional scene. These enhanced depth images are used to perform per-pixel advection, which considers object motion and view changes. Our dual graphics card architecture is able to render the 3D scene at the highest possible frame rate on one graphics card, while doing the depth image warping on a second graphics engine at video refresh rate.

This architecture allows us to compensate for visual artifacts, also called motion judder, arising when the rendering frame rate is lower than the video refresh rate. The evaluation of our method shows motion judder can be effectively removed.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Display Algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality; I.4.8 [Image Processing And Computer Vision]: Scene Analysis—Motion;

Keywords: VR, Motion Estimation, Smooth Motion, Video Refresh Rate, Judder, Dual-GPU, Warping

1 Introduction

Modern Virtual Reality developers use toolkits in order to simplify the implementation and testing of their application. Usually, these toolkits are based on an underlying architcture, which defines how the VR-application behaves in a certain state. Traditional architectures are event driven (eg. [Shaw et al. 1993; Bierbaum et al. 2001; Liere and Mulder 1999]). The application remains in an idle state until an event arrives, which is then processed and may trigger the generation of an image to be drawn on the display. For example, an event can originate from either an interaction device or the simulation process, which in turn may change the state of the 3D scene graph. The renderer will then be required to redraw the scene graph.

The generation rate of events is usually different for each component. The simulation process could run very slowly at 2Hz, while the interaction component produces user input events at 30Hz. Consequently, the scene graph will be rendered at a maximum of 30Hz. However, the refresh rate of the display is typically much higher. For example, 120Hz is a common refresh rate for stereoscopic displays. In this case, the display device will display the same image four times before displaying the next image.

Our motivation for developing a new VR-architecture comes from the observation that visual artifacts, which is often called judder in the video processing community [Marsh 2001], occur if the rendering of a 3D scene is updated at a lower frequency than the video refresh rate of the display device. The perception of judder causes user fatigue and eye-strain. For example, Bles and Wertheim state that: “To avoid headaches and eye strain [...] it is necessary that smooth visual motion will indeed be perceived as smooth.”, furthermore they state that: “When calculations necessary for generating moving images take relatively much time [...] the movements will be seen as consisting of small steps. This is visually quite discomforting.” [Bles and Wertheim 2000].

To overcome the visual artifacts caused by judder, resulting from a mismatch between rendering and video frame rate, a VR system should not be driven by the update rates of the simulation, rendering or input devices. Rather, the video rate of the display should be the driving factor.

In this paper, we introduce the design, implementation and evaluation of a VR-architecture for smooth motion. Forward depth image warping using predictive per-pixel advection is used to estimate new images. In addition to color and depth, our approach records per-pixel motion information during rendering of the three-dimensional scene. These enhanced depth images are used to compute predicted intermediate images. The architecture allows us to compensate for object motion in addition to view point motion.

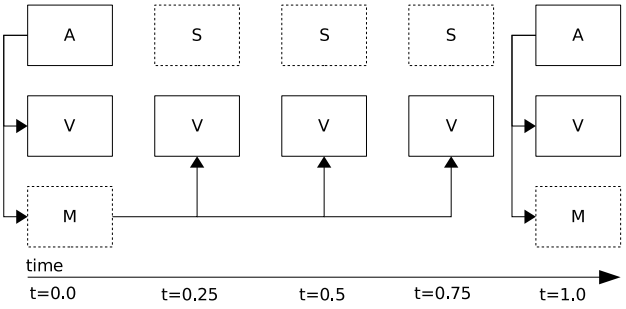

Figure 2: *The client produces application frames (A), and corresponding motion fields (M). The display device displays video frames (V) in sequence. Several video frames are displayed before the next application frame is generated. The in-between application frames that should have been displayed had the client been fast enough can be simulated (S). Furthermore, the server uses the initial application frame (A) and the motion field (M) to generate extrapolated video frames (V). The error in the motion estimation and extrapolation shows as the difference between the simulated (S) and displayed (V) frames.*

The architecture is implemented using dual graphics cards. Three-dimensional scenes are rendered at the highest possible frame rate on one graphics card, while depth image warping is performed on a second graphics card at video refresh rates. This approach is illustrated in Figure 1.

2 Architecture

The goal of the architecture is to draw new video frames on the display with a fixed, constant frame rate, regardless of the rate at which the application is producing new application frames. Throughout this section we assume a display refresh rate of 120Hz; therefore, video frames must be rendered at a constant rate of 120Hz. When the rendering of a new application frame takes longer than 1/120th of a second, it is impossible to sequentially generate application frames and render consecutive video frames at 120Hz. Consequently, some form of parallel processing has to be used to generate these frames at the same time.

Current video hardware, while internally highly parallel in nature, is ill-suited to perform high-level parallel tasks. While it is possible to run two high-level parallel processes on the same GPU, there is a shared context state on the GPU that must be carefully maintained. These context switches of the video hardware are typically arranged for by the operating system and are very expensive. Furthermore, once a command has been issued to the GPU it will run until completion without interruption. Therefore, it would be very difficult, if not impossible, to implement parallel operations such as synchronization primitives and, more importantly, priority-based process scheduling. Without prioritized time slicing on the GPU, there is no way to guarantee our video frames will always be drawn at a fixed rate.

To overcome the problems of running two parallel processes on the same GPU, our architecture uses two completely independent video cards. Both cards are connected to the same PCIe bus using a dual GPU mainboard. Communication between the GPUs is achieved through shared system memory over the PCIe bus. In order to communicate frame-buffer information from the first to the second GPU, a circular producer/consumer buffer is realized in the shared memory. The *client* process issues an application frame to be rendered by the first GPU, which is then written to an empty

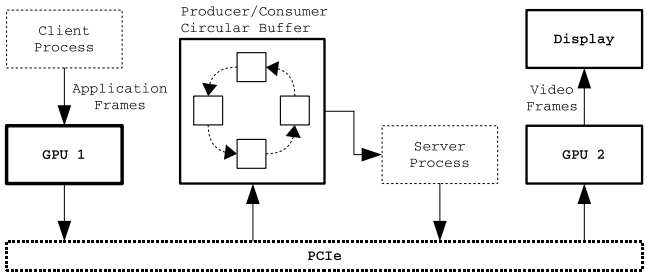

Figure 3: *The constant video frame rate VR-architecture. Data typically flows in the following sequence: Application, GPU1, PCIe, Shared memory, PCIe, GPU2, Display. By using two distinct GPUs it is possible to render application frames and video frames in parallel at different rates. The client process produces application frames into the circular buffer using GPU 1, which in turn are consumed by the server process using GPU 2.*

slot in the circular buffer. The *server* process polls the buffer in a non-blocking fashion whether a new frame is available. If this is the case, the filled slot is copied to the second GPU and released again. This is illustrated in Figure 2 and Figure 3.

3 Motion Estimation and Extrapolation

3.1 Client Implementation

In order to generate a per-pixel motion field, every geometric object stores a transformation matrix for the previous, as well as the current application frame. When an object is rendered, both the current and previous vertex positions are interpolated in the pixel shader. In this way, two 3D positions are available for each pixel: the current position P^n and the previous position P^{n-1}. The position of the pixel for the next application frame can now be predicted according to the expression $P^{n+1} = P^n + V^n$, where $V^n = P^n - P^{n-1}$ is the 3D motion vector.

The client renders both the pixel color data and the 3D motion vectors to two separate buffers using GPU frame buffer objects (FBOs) and multiple render target (MRT-2) functionality. The buffers consist of 4-vectors of 8-bit bytes for each pixel color, and 32-bit floats for the motion data. For the color of the pixel we store the red, green and blue components and use the alpha channel as a flag to indicate the pixel has a valid motion vector. The background is cleared with an alpha value of zero. The three components of the motion vector V^n are stored in the first three components of the second 4-vector. The actual depth of the pixel is stored in the last component, so it is possible to reconstruct the 3D position using the screen coordinates of a pixel and this depth value. This results in the vectors $(P_r^n, P_g^n, P_b^n, 1)$ and $(V_x^n, V_y^n, V_z^n, P_z^n)$.

3.2 Server Implementation

The server is responsible for drawing video frames at the fixed video frame rate. As applications frames are typically generated at a lower rate than the video frame rate, the server must perform frame extrapolation. Suppose the video frame rate is fixed at 120Hz with an application frame rate of 40Hz, in this case there will be a 1:3 ratio between application and video frames. So for every received application frame, the server needs to generate two extrapolated video frames along with the received application frame.

When the server finds that a new application frame is available in the producer/consumer buffer, it starts by converting it to a 3D

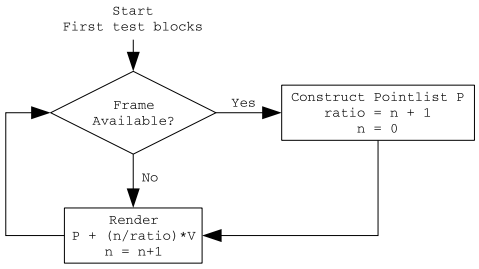

Figure 4: *This figure shows the operational flow chart for the server. The ratio between application and video frames is assumed to be equal to the last cycle. Whenever an application frame is not available, the previously generated point list P is displaced by a fraction of the motion vectors V, dependent on the ratio and the number of frames n that were already missed previously.*

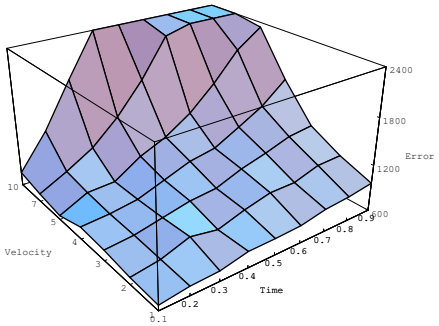

Figure 6: *The absolute number of error pixels for extrapolated video frames at time t for varying velocities. The ratio between application and video frames is 1:10. The display resolution was 1024x768 pixels. The velocity axis represents degrees/frame, and the time axis shows the nine extrapolated video frames.*

point-list residing at server memory. The construction of the point-list is currently done on the CPU, but this could be converted to a GPU implementation in the future. The process iterates over the entire grid of 2D pixels, and for every pixel a test is made whether the alpha value is 1, indicating a valid motion vector. For each pixel passing the alpha test, the 2D grid coordinates are converted into camera near-plane coordinates in the range $[-0.5, 0.5]^2$. Next, the actual 3D point is reconstructed by making use of the stored P_z^n value residing in the motion data and the known virtual camera parameters. This can be realized with the expression $P_{xy}^n = (W_{xy}/f) \cdot (P_z^n P_{xy}^n)$ where W_{xy} is the focal plane width and height, and f the camera focal length. The point-list now stores the actual 3D position P_{xyz}^n, the 3D motion vector V_{xyz}^n and the pixel color P_{rgb}^n.

Next, the point cloud P_{xyz}^n is rendered by the second GPU. The rendering of the point cloud is is achieved by transmitting all the position, motion and color data over the PCIe bus to the second GPU using vertex buffer object (VBO) functionality. In this way, the data remains in video hardware memory. Finally, a hardware vertex program fetches the correct position and color for each point and renders it as normal.

When a new application frame is not available, an extrapolation of the previous data is performed. The server keeps track of the observed ratio r between application and video frames, and also the number of times n an application frame was not available since the last received frame. Given this data, the server can extrapolate the positions P_{xyz}^n in the previously stored point-list by adding a fraction of the stored motion vector V_{xyz}^n. The scale factor is given by $\frac{n}{r}$. This is shown schematically in Figure 4. After the correct scale factor for the motion vector has been determined, the points can be rendered as a point cloud.

4 Results

We have implemented the architecture using an NVidia GeForce 8800 GTX, and an NVidia Quadro FX5600 graphics card. The graphics cards are connected over the PCIe bus. The system uses an AMD Athlon-64 X2 3800+ dual core processor, therefore the client and server process can run on a separate core. An Iiyama Vision Master Pro 512 22 inch CRT monitor operating at 120Hz refresh rate was used for display.

In order to obtain results we rendered a simple scene at a resolution of 1024x768 consisting of a concave object as is shown in Figure 5. The object is rotating along two of its axes simultaneously, resulting in non-linear motion on the camera plane. The speed of rotation is expressed in degrees per application frame. We have used a simple scene so the effects of motion extrapolation are clearly visible.

To estimate the quality of the extrapolated video frames produced by our implementation, we have used a simulation program. A ratio of 1:10 between application and video frames was artificially maintained, simulating a 12Hz update rate at a 120Hz display. This means that for every application frame 10 video frames are generated, nine of which are extrapolated from the application frame and its associated motion field. The first application frame is generated at $t = 0.0$, and the next consecutive application frames are generated at $t = 1.0$, $t = 2.0$, etc. The nine extrapolated video frames from the application frame at $t = 0.0$ correspond to animation timings $t = 0.1$, $t = 0.2$ ... $t = 0.9$. As a basis for comparison, the simulation program also renders the application frames that should be displayed at times $t = 0.1$, $t = 0.2$ ... $t = 0.9$. The quality of the motion estimation and extrapolation can be inspected by comparing the simulated application frames and the extrapolated video frames. For example, the first extrapolated video frame occurs at $t = 0.1$ and is generated from the application frame and motion data at $t = 0.0$. Next, we also render a simulated application frame for $t = 0.1$ and compare it with the extrapolated video frame at $t = 0.1$.

The simulated application frames and the extrapolated video frames were compared per pixel to asses the quality of extrapolation. Whenever two corresponding pixels differ for any color channel the pixel is marked as an error pixel. A small threshold value for the difference is used to avoid marking small, imperceivable deviations in pixel color as errors. The total amount of error pixels is used as a quantitative measure for the quality of the extrapolated video frame. The errors are visualized by rendering error pixels in red over a grayscale luminance version of the simulated application frame. This is illustrated in Figure 5.

From Figure 5, we can distinguish between three types of errors. First, there is the error due to occlusion, which is present near the the yellow sphere. Occlusion errors occur when in the original application frame a part of the scene is occluded by other geometry. Therefore, there are no points present in the point cloud representing that part of the scene, and thus these regions are never reached by any point after motion extrapolation. The second type of error is caused by errors in motion estimation. If the added motion vectors become larger, the linear approximation of motion deviates more from the actual motion. This can be seen clearly at the edges of the object. Finally, the third type of error is due to aliasing or gaps. The presence of gaps is reduced by the use of resolution scaling,

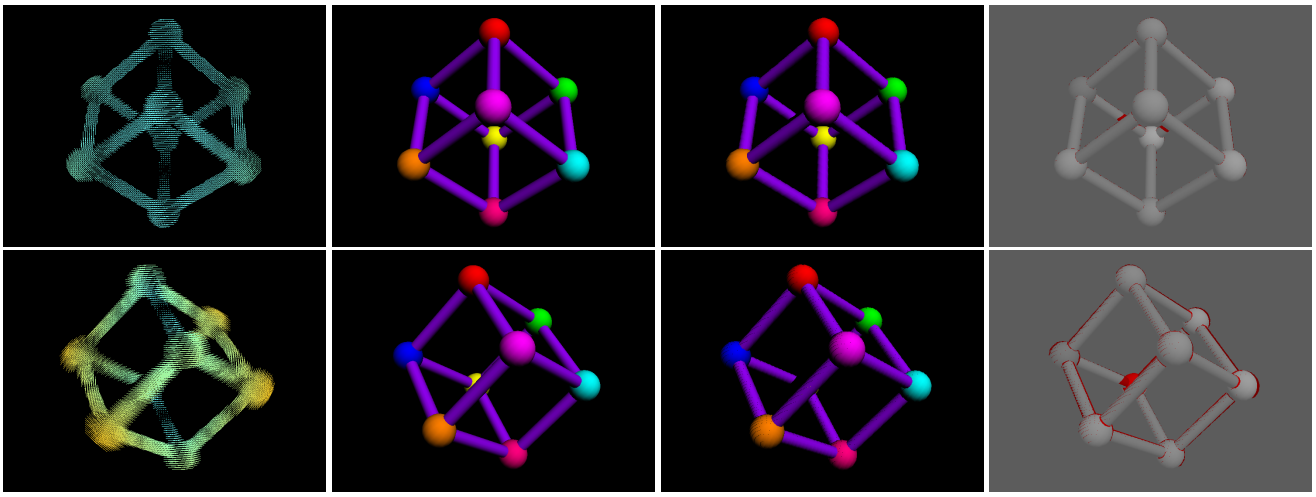

Figure 5: *From left to right: the motion field associated with the actual application frame at $t = 0$, the simulated application frame ahead in time, the video frame extrapolated from the application frame and the motion field, and finally the visualization of the error between the extrapolated video frame and the simulated application frame. The ratio between application and video frames is 1:10. The top row shows these images for a velocity of 4 degrees/frame at time $t = 0.6$, i.e. the 6th extrapolated video frame. The bottom row shows a velocity of 10 degrees/frame at time $t = 0.7$.*

however if motion becomes large this is no longer sufficient. This type of error is visible on the orange sphere among other locations.

Figure 6 show the number of pixel errors for each of the nine extrapolated video frames at different velocities. It can be seen that the error increases as the time of the extrapolated video frame increases and if the rotational velocity increases. This shows that the error in motion extrapolation is proportional to the length of the added motion vectors as expected. Therefore, different ratios between application and video frames are expected to produce similar results. For example, a 1:4 ratio is expected to result in a maximum error close to the observed error for time $t = 0.3$ in the case of the 1:10 ratio.

For the 500k polygon scene shown in Figure 1 our server implementation was able to maintain a steady 120Hz video frame rate for a resolution of 320x200. For a resolution of 640x480 the server was no longer able to maintain the required 120Hz frame rate and dropped to 60Hz.

5 Discussion

We presented the design, implementation and evaluation of a VR-architecture for smooth motion. The architecure can be benifical to those VR applications in which the simulation or rendering rate is lower than the video refresh rate.

We briefly discuss various shortcomings of our implementation. First, the performance of the system is not adaquate for typical resolutions found in VR displays. Currently, a 120Hz video frame rate for a resolution of only 320x200 can be met. There are two reasons for this. The implementation transfers data from one video card to the other through shared memory and the PCIe bus. For larger resolutions this is a limiting factor, as the throughput over the PCIe bus is no longer sufficient. It is therefore desirable to be able to copy data directly between the video cards. Unfortunately, current generation video cards do not provide this functionality. At the server side, the point-list is generated completely on the CPU. An alternative approach would be to generate the point list on the GPU, for example by the use of histogram pyramids [Ziegler et al. 2006].

Our motion estimation and extrapolation method is but one of many choices. Linear extrapolation is a reasonable estimate as we are dealing with very short time intervals during which the motion is typically limited. However, in order to reduce potential errors, more advanced predictive filtering methods, such as the Kalman filter, can be applied instead.

A fundamental problem with rendering displaced point clouds is the potential occurrence of gaps, i.e. 2D output pixels that are never reached by a displaced 3D point. In order to avoid such gaps, a 2D mesh of connected quads can be constructed from the point cloud, although care must be taken since the rendering may introduce other artifacts due to a self-intersecting mesh.

References

BIERBAUM, A., JUST, C., HARTLING, P., MEINERT, K., BAKER, A., AND CRUZ-NEIRA, C. 2001. Vr juggler: A virtual platform for virtual reality application development. In *VR*, 89–96.

BLES, W., AND WERTHEIM, A. 2000. Appropriate use of virtual environments to minimise motion sickness. *RTO MP58*, 7.1–7.9.

LIERE, R., AND MULDER, J. 1999. PVR - an architecture for portable VR applications. In *EGVE '99 Conference Proceedings*, 125–135.

MARSH, D. 2001. Temporal rate conversion. *Microsoft archived paper: http://www.microsoft.com/whdc/archive/TempRate.mspx*.

SHAW, C., GREEN, M., LIANG, J., AND SUN, Y. 1993. Decoupled simulation in virtual reality with the MR toolkit. *Information Systems 11*, 3, 287–317.

ZIEGLER, G., TEVS, A., THEOBALT, C., AND SEIDEL, H.-P. 2006. On-the-fly point clouds through histogram pyramids. In *11th International Fall Workshop on Vision, Modeling and Visualization 2006 (VMV2006)*, Eurographics, 137–144.