Real-time Simulation of Motion-based Camera Disturbances

Dennis Pachur[1], Tim Laue[2], and Thomas Röfer[2]

[1] Fachbereich 3 - Mathematik und Informatik, Universität Bremen,
Postfach 330 440, 28334 Bremen, Germany
E-Mail: pachur@informatik.uni-bremen.de
[2] Deutsches Forschungszentrum für Künstliche Intelligenz GmbH,
Sichere Kognitive Systeme, Enrique-Schmidt-Str. 5, 28359 Bremen, Germany
E-Mail: {Tim.Laue,Thomas.Roefer}@dfki.de

Abstract. In the RoboCup domain, many robot systems use low-cost image sensors to perceive the robot's environment and to locate the robot in its environment. The image processing typically has to handle image distortions such as motion blur, noise, and the properties of the shutter mechanism. If a simulator is used in the development of the control software, the simulation has to take account for these artifacts. Otherwise, the performance of the image processing system in the simulation may not correspond to its performance on the real robot; it may even perform worse.

The effect of motion blur has been widely used for special effects both for movies and for computer games. While real-time algorithms using modern graphics hardware came up in recent years, the image distortion resulting from a so-called rolling shutter has not been in focus so far. In fact, this effect is not relevant for gaming, but it is for simulating low-cost cameras of robots.

In this paper, we present an efficient way to simulate the rolling shutter effect using per-pixel velocities. In addition, we improve the velocity buffer method for creating motion blur using the current speed of each pixel in real-time. The application of our approach is shown exemplarily for the head-mounted camera of a humanoid soccer robot.

1 Introduction

The quality of a robot simulation is inter alia determined by its ability to calculate realistic measurements for the sensors simulated. Each simulation can only approximate reality up to a certain level of detail. For many applications, the missing details are not relevant and the interaction between the robot software and the simulated world is not affected. Nevertheless, the characteristics of some sensors matter to an extent that leads to a significant mismatch between simulation and reality. Such a mismatch is relevant in two cases: the first and naïve case is that the robot control software is initially developed only in a simulator, and it does not handle sensor distortions that are not present in the simulation, but surely will be in reality. In the other case, the more experienced developer

Fig. 1. Examples of image distortion caused by the rotation of a CMOS image sensor with a rolling shutter.

would develop the software on the real robot and in simulation side by side. However, software tuned for real sensor readings may perform worse or even fail if confronted with unrealistically simple sensor readings coming from a simulator, rendering the simulator as a tool rather useless. Hence, it is important that a simulation generates all the sensor distortions that are actively handled by the robot control software developed.

Figure 1 shows examples of typical distortions, using images from a CMOS camera that is part of the head of a humanoid robot. While the blurry impression is caused by motion blur, the distortion that bends and squeezes the yellow goal is produced by the so-called rolling shutter. Instead of taking images at a certain point in time, a rolling shutter takes an image pixel by pixel, row by row. Thus the last pixel of an image is taken significantly later than the first one. If the camera is moved, this results in image distortions as shown in figure 1. Both artifacts, motion blur and rolling shutter effect, can be generalized as temporal aliasing effects caused by the light integration of the image sensor. While the blurriness of a pixel results from the movement in the scene while the pixel is exposed, the rolling shutter causes the pixels of an array to be exposed in a serialized way.

Compensating this effect becomes significantly important when working with robots which move their cameras fast or operate in a rapidly changing environment. In the RoboCup domain, this particularly affects robots with pan-tilt heads, e. g. in the Four-legged and the Humanoid league. To overcome problems regarding world modeling, which obviously arise from these disturbances, different compensation methods have been developed, e. g. by Nicklin et al. [1] or by Röfer [2]. A necessity of simulating these disturbances can be derived from this explicit handling by the software of different teams.

Within the RoboCup community, a variety of robot simulators is currently used. Among the most advanced and established ones are, for example, Webots [3], Microsoft Robotics Studio [4], and the USARSim [5]. In general, they are able to simulate camera images at a high level of detail, e. g., by simulating

lights and shadows. But so far, none of them addresses the problem of image disturbances.

The contribution of this paper is an efficient approach for simulating common image disturbances, i. e. the rolling shutter effect and motion blur, in real-time. This approach has been implemented for an existing robot simulator; a movable camera of a simulated humanoid robot is used as an example application.

The paper is organized as follows: Section 2 provides an overview of related work about generating image disturbances. In Section 3, our approach is presented, follow by the experimental results in Section 4.

2 Related Work

This section gives a short overview of existing models and methods for image distortion. Considering performance as well as their capabilities, design choices for our approach are derived.

2.1 Image Distortion Model

In [6], the image distortion caused by a rolling shutter within a CMOS image sensor is given by the pixel position $\boldsymbol{x}$ at a certain time t. $\boldsymbol{x}_t^u$ is the undistorted position of a pixel at time t and $\boldsymbol{u}_t$ is its motion. By taking into account the motion since the start of the acquisition of the current image, the position is transformed by the motion b_t within the interval $[0, t_{\boldsymbol{x}}]$. The time of the pixel exposure at t_a can be calculated by

$$t_{\boldsymbol{x}} \approx \frac{\boldsymbol{x}_{x,t}^u}{WH} + \frac{\boldsymbol{x}_{y,t}^u}{H} \tag{1}$$

with W and H being the width and the height of the pixel array, and $\boldsymbol{x}_{x,t}^u$ and $\boldsymbol{x}_{y,t}^u$ being the coordinates of the pixel in the undistorted image. The distorted pixel position $\boldsymbol{x}_t^d$ can then be calculated by equation 2:

$$\boldsymbol{x}_t^d = \boldsymbol{x}_t^u + \int_{t=0}^{t_{\boldsymbol{x}}} \boldsymbol{u}_t dt \tag{2}$$

The motion during image acquisition is assumed to be constant.

For using this equation for distorting an image, the velocity has to be calculated. Whereas camera rotations cause a homogeneous velocity vector field for the complete image area, camera translations and object movements cause local velocities. To take these movements into account, per-pixel velocities have to be determined. This is common practice for a class of algorithms generating motion blur.

2.2 Integrating Motion Information into Rendered Images

Computer-generated images can be produced either in consideration of the image quality or of the time needed for the generation. The algorithms applied to calculate real-time graphics are based on complex formulas containing approximating parts. For the field of motion information (e. g. motion blur), real-time algorithms commonly approximate the visibility function. This results in generating artifacts in image areas with little spatial information. A complete equation without approximation is given in [7] but can only be used by a ray-tracer which calculates one pixel at once. Real-time rendering engines, such as OpenGL, use a rasterizer to approximate the spatial information by constructing surfaces between points and projecting the rasterized surface points to the image plane by interpolating the information along the edges.

For real-time graphics, a few classes of algorithms have made their way into everyday use: the accumulation of images from different points in time within the light integration interval [8], the extension of the object geometries in direction of the object's motion direction [9, 10], and the post processing of the image using velocities calculated previously for either the objects [11] or the image pixels [12, 13]. Due to the scene-independent performance of post processing, algorithms arose for iterative applications [14] that can create motion-blurred images with almost no artifacts.

Such methods are also used in the work presented here since the applied image distortion model is based on post-processing the image using per-pixel velocities. Hence motion-blur and the rolling shutter effect can be integrated without any additional pre-calculations.

2.3 Calculating Per-Pixel Velocities

The velocity $v_{\boldsymbol{x}}$ of a pixel $\boldsymbol{x}$ is given by the difference of the projections dP of a point at two time steps $t, t-1$ and can be calculated as in equation 3.

$$\boldsymbol{v_x} = \frac{dP}{dt} \qquad \text{with } dP = P_t - P_{t-1} \tag{3}$$

Therefore, the matrix stack from modelview to projection for both time steps must be known. For image generation, this is implicitly known for each vertex sent through the rendering pipeline. To generate velocities for each pixel, the difference calculated for each vertex can be interpolated over the adjacent polygons by varying an attribute while rasterizing the surface. By using up-to-date hardware, this velocity vector field can be rendered into an image using multiple render targets, requiring a single rendering pass only. The result can be copied to a texture and be stored for later use if more than a single velocity image is needed within the motion blur appliance and the render target is not chosen as a texture anyway.

A more recent method to calculate the per-pixel velocity is described in [15]. The velocities are determined by taking the matrices of both time steps and

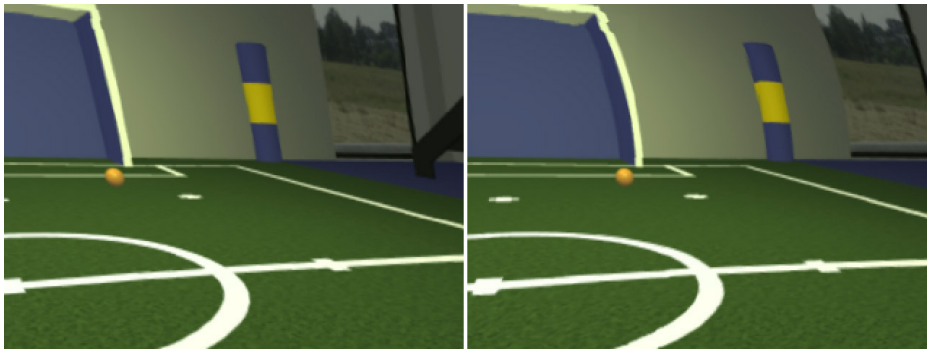

Fig. 2. Examples of image distortion for a rolling shutter simulation based on per-pixel velocity. The left image has been generated using a constant velocity, whereas interpolated velocities have been applied to the right image.

reconstructing a three-dimensional image point through employing the depth image commonly used by the rasterizer to solve the visibility function.

Once the per-pixel velocities have been calculated, the image can be distorted by using a per-pixel operation. Nowadays, this operation can be accelerated using an image-aligned plane, a feature of a modern fragment shader, and a variable shading language such as CG, GLSL or HLSL.

3 Simulating Image Disturbances

Our simulator SimRobot [16] uses OpenGL to visualize the scene and to generate camera images. Hence, we use frame buffer objects to render the image data and the per-pixel velocities in only a single pass. The rendering system supports antialiasing and render-target-switching to obtain high visual quality and to store up to six velocity images for later use. The motion blur approach of [13] was chosen to blur the image using multiple velocity buffers.

3.1 Calculating Per-pixel Velocities

For a single image pixel $\boldsymbol{x}$, the fragment shader determines the pixel position offset by interpolating the velocities of the last two velocity buffer images $\boldsymbol{v}_{\boldsymbol{x}}^{curr}$ and $\boldsymbol{v}_{\boldsymbol{x}}^{prev}$. If the image resolution is known, the interpolated velocity $\boldsymbol{v}_{\boldsymbol{x}}$ of a pixel $\boldsymbol{x}$ can be calculated using

$$\boldsymbol{v}_{\boldsymbol{x}} = f\boldsymbol{v}_{\boldsymbol{x}}^{curr} + (1-f)\boldsymbol{v}_{\boldsymbol{x}}^{prev} \tag{4}$$

with

$$f = \frac{\boldsymbol{x}_x}{WH} + \frac{\boldsymbol{x}_y}{H}. \tag{5}$$

For simplifying the image distortion with negligible imprecision, the temporal offset caused by the horizontal position $\boldsymbol{x}_x$ in the image can be discarded. Thus

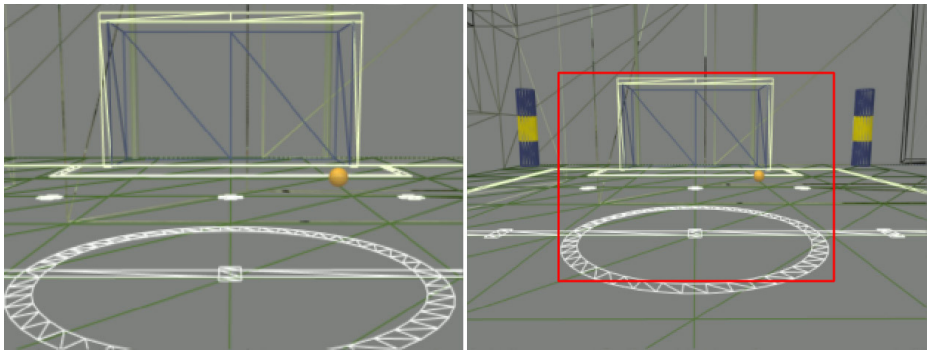

Fig. 3. The left image shows the part of the model that should be rendered. The right image shows the extended field of view used to be able to determine the pixel velocities correctly close to the image border.

a fragment shader can use texture coordinates to obtain generalized image coordinates in the range of $[0 \dots 1]$ that can be applied to any image resolution. In this case, the position of the pixel in the image can be simply calculated using

$$f = \boldsymbol{x}_y \qquad y \in [0 \dots 1] \tag{6}$$

The interpolated velocity adds information about the change of speed. If the camera starts rotating within the time of exposure, the bottom left pixel will receive most of the speed of the velocity calculated previously while the upper right pixel will almost use the current velocity. This facilitates not only creating the image distortion but also adjusting the motion blur itself.

To adjust the pixel position $\boldsymbol{x}$ in the local image, equation 2 is used. The coordinates of the current pixel $\boldsymbol{x}$ have to be distorted by the calculated velocity.

$$\boldsymbol{x}^d = \boldsymbol{x} - f\boldsymbol{v}_{\boldsymbol{x}} \tag{7}$$

Figure 2 shows two images generated by this method. While the right image uses interpolated velocities and identifies the change of rotation speed of the camera, the left image only uses the current velocity $\boldsymbol{v}_{\boldsymbol{x}}^{curr}$ as distortion base.

3.2 Reducing Image Artifacts

The images in Figure 2 show distortion artifacts at the top left corner of the goal. They result from calculating per-pixel velocities for a rasterized image as mentioned before.

To minimize image artifacts, the represented time integral can be adjusted and be shifted slightly into the future. This can be done by determining the new pixel position $\boldsymbol{x}^d$ with $f' = f - 0.5$. The center of the distortion is shifted from the top of the image to its center resulting in a negative distortion direction in the bottom half and a positive distortion direction in the top half of the image.

The shorter distortion vectors result in fewer artifacts. If the change of speed is too fast within two time steps, the bottom half starts to seem swinging as the prediction of the upcoming motion is failing.

When the camera is moving, a subset of the distorted image points is located outside the image area. To avoid leaving the image space (and thus leaving points outside the plane clear or repeating the image at its borders) the render system has to provide more than the usual visual image field. By adjusting the field of view before rendering the scene and correcting the texture coordinates afterwards, the area of the working image can be extended (cf. Fig. 3).

3.3 Applying Motion Blur

For the new pixel location $\boldsymbol{x}^d$, another velocity has to be calculated to determine the locally distorted velocity for applying the motion blur itself. Again, using interpolated velocities will result into more image information. It has to be mentioned that the interpolation method for the velocity buffers determines the change of motion in the final image. Hence, using more buffers will result in a better approximation.

As known from other pixel motion blur methods, colors are accumulated and divided by the number of accumulated samples along the vector of the velocity image. The number of samples can be determined by the length of this vector resulting in a higher rendering speed when there is little motion. If interpolated velocities are used, the blur scales correctly with the change of speed. Figure 4 shows various images of the combined usage of interpolated velocities for image distortion and blurring. In the upper left image, nearly no movement is present at the bottom line, resulting in less distortion and blur distance. At the upper end of the image, both distortion and blur are present. The upper right image shows a camera movement towards the goal. The camera speed is decreased and stops within the integration interval.

3.4 Noise

To simulate CMOS image sensors, our simulator uses the previously presented techniques in conjunction with dynamically generated noise. The noise model is based on variances for each color channel taken by difference images of an image sensor chosen. The model tries to adopt the static, color, and edge noise but fails on the dynamic temporal behavior of the noise. Therefore only the noise generated for a single image shows nearly correct statistics and the corresponding visual effect. This calculation is part of a post process of the rendering and it only marginally affects the rendering time. Figure 5 shows a real camera image in the top left corner and a contrast enhanced difference image to another image of the same scene in the middle. The histograms in the upper right corner show the color channel variances of the unenhanced difference image. In comparison, the lower row shows a simulated static scene with the image noise generated. The noise helps to create inhomogeneous color areas in the image, which also appear in real camera images.

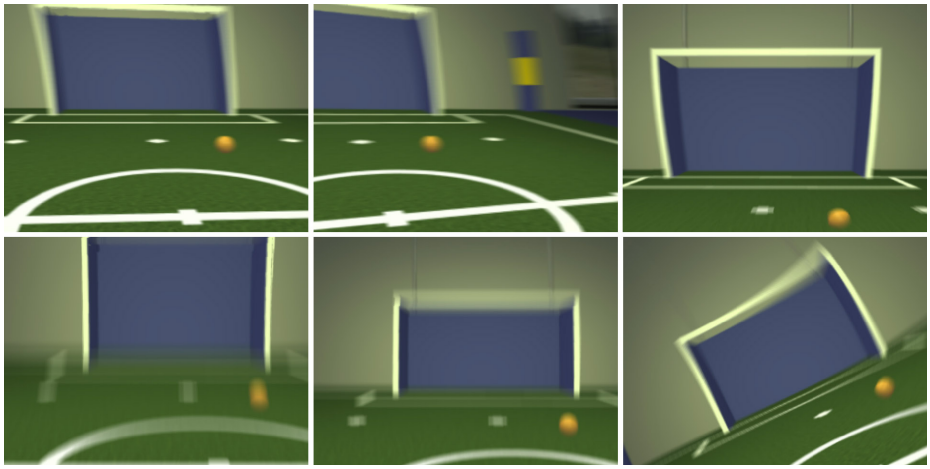

Fig. 4. Combining interpolated velocities used for the rolling shutter distortion with a commonly used motion blur method.

Table 1. Average rendering times. Times not measured are blanked.

	1 Sample	6 Samples	24 Samples	240 Samples
Accumulation Buffer	6.632ms	39.114ms	151.424ms	-
Velocity Buffer	-	13.436ms	14.372ms	15.153ms
Generic Distortion Model	-	-	-	16.266ms

4 Results

The generic distortion model allows a variety of different image effects. In addition, it can simply extend an integrated temporal antialiasing algorithm based on per-pixel velocities with nearly no noticeable performance change on contemporary hardware. For comparison, we implemented the relevant state-of-the-art motion blur methods and measured the rendering times in milliseconds. The benchmark scene contains 73 different objects from a simple box up to a complete room. Per frame, about 12000 triangles are rendered. In addition, the scene uses five light sources. The object lighting is calculated per fragment. Every image is rendered using 16 samples for smooth edges. Table 1 shows averaged times taken on an up-to-date desktop PC (AMD Athlon 64 X2 Dual Core, 2.4 GHz, NVIDIA GeForce 8800 GTX).

While the accumulation buffer method only handles the motion blur of a scene, the default velocity buffer method using two buffers shows better performance. It allows the integration of other post processes without loosing performance, as it has been done for the work presented here. The time needed for an accumulated image increases linearly with the number of images rendered. The time for the double buffered velocity method depends on the performance of the graphics hardware. The additional calculations performed by the fragment

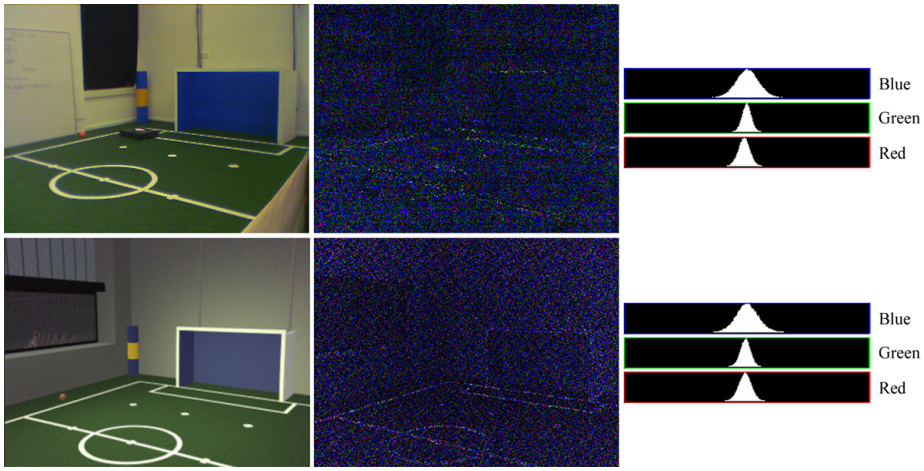

Fig. 5. Comparison of a real camera image at the upper left and its simulated equivalent image at the bottom right. The middle column shows the corresponding contrast enhanced difference images of the noise. The third column shows normalized histograms per color channel.

shader units have a low impact on the rendering time. Hence, our extensions made to realize image distortions and noise do not significantly impede the performance.

Finally, Figure 6 compares real images on the left with their simulated counterparts on the right. The top row shows a static scene while the bottom row shows an image from a rotating camera.

5 Conclusion

In this paper, we presented an approach that is able to simulate common image disturbances, i. e. motion blur and the rolling shutter effect. Especially the latter might influence the robot's perception in a way that it has to be handled explicitly within the vision software. Thus its simulation is necessary to allow a more realistic testing within a simulation environment. The quality of our approach is depicted in Figure 6.

Contemporary graphics hardware already provides mechanisms that allow an efficient implementation of this simulation, although originally being developed especially for gaming purposes. The performance measurements presented in Table 1 show that the approach is capable of operating in real-time, i. e. to generate distorted images at a frame rate of up to 60 Hz. This could be assumed to be satisfactory for most applications.

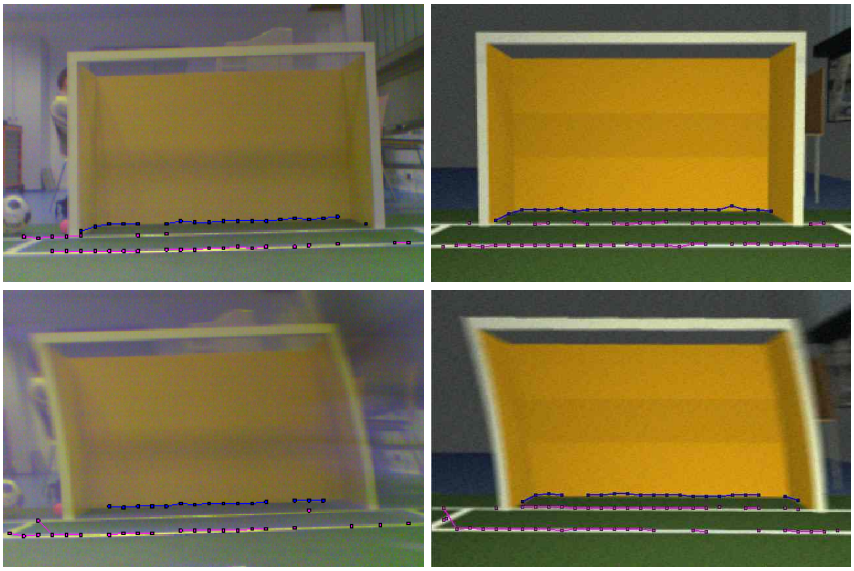

Fig. 6. Comparison of real images on the left and equivalent simulated images on the right. The dots and lines represent percepts of the robot's vision system.

Acknowledgements

This work has been partially funded by the Deutsche Forschungsgemeinschaft in the context of the Schwerpunktprogramm 1125 (*Kooperierende Teams mobiler Roboter in dynamischen Umgebungen*).

References

1. Nicklin, S.P., Fisher, R.D., Middleton, R.H.: Rolling shutter image compensation. In Lakemeyer, G., Sklar, E., Sorrenti, D., Takahashi, T., eds.: RoboCup 2006: Robot Soccer World Cup X. Lecture Notes in Artificial Intelligence, Springer (2007)
2. Röfer, T.: Region-based segmentation with ambiguous color classes and 2-D motion compensation. In Visser, U., Ribeiro, F., Ohashi, T., Dellaert, F., eds.: RoboCup 2007: Robot Soccer World Cup XI. Lecture Notes in Artificial Intelligence, Springer; http://www.springer.de/
3. Michel, O.: Cyberbotics Ltd. - WebotsTM: Professional mobile robot simulation. **1**(1) (2004) 39–42
4. Jackson, J.: Microsoft Robotics Studio: A technical introduction. Robotics and Automation Magazine **14**(4) (2007) 82–87
5. Wang, J., Lewis, M., Gennari, J.: USAR: A game based simulation for teleoperation. In: Proceedings of the 47th Annual Meeting of the Human Factors and Ergonomics Society. (2003)
6. Cho, W.h., Kim, D.W., Hong, K.S.: CMOS digital image stabilization. IEEE Transactions on Consumer Electronics (8 2007)

7. Sung, K., Pearce, A., Wang, C.: Spatial-temporal antialiasing. IEEE Transactions on Visualization and Computer Graphics **8**(2) (4 - 6 2002) 144 – 153
8. Haeberli, P., Akeley, K.: The accumulation buffer: Hardware support for high-quality rendering. ACM SIGGRAPH Computer Graphics **24**(4) (8 1990) 309–318
9. Tatarchuk, N., Brennan, C., Isidoro, J.: Motion blur using geometry and shading distortion. In: ShaderX2, Shader Programming Tips and Tricks with DirectX 9. Wordware Publishing, Inc. (10 2003) 299 – 308
10. Jones, N.E.: Real-time geometric motion-blur for a deforming polygonal mesh. Master's thesis, Texas A&M University, Heatherdale, Houston, Texas (5 2004)
11. Nelson, M.L., Lerner, D.M.: A two-and-a-half-D motion-blur algorithm. ACM SIGGRAPH Computer Graphics **19**(3) (7 1985) 85–93
12. Green, S.: Stupid OpenGL shader tricks. Game Development Conference (2003) 3–16
13. Microsoft Corporation: Pixelmotionblur sample (2005) MSDN Microsoft Developer Network DirectX SDK Sample, http://msdn2.microsoft.com/en-us/library/bb147267.aspx, visited 15.07.07.
14. Shimizu, C., Shesh, A., Chen, B.: Hardware accelerated motion blur generation. Technical Report 2003-01, University of Minnesota Computer Science Department, Twin Cities, Minnesota (2003)
15. Rosado, G.: Motion blur as a post-processing effect. In Nguyen, H., ed.: GPU Gems 3. Pearson Education, Inc., Bosten, Massachusetts (7 2007) 575–581
16. Laue, T., Spiess, K., Röfer, T.: SimRobot - A General Physical Robot Simulator and Its Application in RoboCup. In Bredenfeld, A., Jacoff, A., Noda, I., Takahashi, Y., eds.: RoboCup 2005: Robot Soccer World Cup IX. Number 4020 in Lecture Notes in Artificial Intelligence, Springer; http://www.springer.de/ (2006) 173–183