Handling Photographic Imperfections and Aliasing in Augmented Reality

Jan Fischer, Dirk Bartz

WSI-2006-03
Juli 2006

Graphisch-Interaktive Systeme
Wilhelm-Schickard-Institut
Universität Tübingen
D-72076 Tübingen, Germany
e-mail: `fischer@gris.uni-tuebingen.de`
WWW: `http://www.gris.uni-tuebingen.de`

ISSN 0946-3852

Handling Photographic Imperfections and Aliasing in Augmented Reality

Jan Fischer* Dirk Bartz

WSI/GRIS - VCM, University of Tübingen

Abstract

In video see-through augmented reality, virtual objects are overlaid over images delivered by a digital video camera. One particular problem of this image mixing process is the fact that the visual appearance of the computer-generated graphics differs strongly from the real background image. In typical augmented reality systems, standard real-time rendering techniques are used for displaying virtual objects. These fast, but relatively simplistic methods create an artificial, almost "plastic-like" look for the graphical elements.

In this paper, methods for incorporating two particular camera image effects in virtual overlays are described. The first effect is camera image noise, which is contained in the data delivered by the CCD chip used for capturing the real scene. The second effect is motion blur, which is caused by the temporal integration of color intensities on the CCD chip during fast movements of the camera or observed objects, resulting in a blurred camera image. Graphical objects rendered with standard methods neither contain image noise nor motion blur. This is one of the factors which makes the virtual objects stand out from the camera image and contributes to the perceptual difference between real and virtual scene elements.

Here, approaches for mimicking both camera image noise and motion blur in the graphical representation of virtual objects are proposed. An algorithm for generating a realistic imitation of image noise based on a camera calibration step is described. A rendering method which produces motion blur according to the current camera movement is presented. As a by-product of the described rendering pipeline, it becomes possible to perform a smooth blending between virtual objects and the camera image at their boundary. An implementation of the new rendering methods for virtual objects is described, which utilizes the programmability of modern graphics processing units (GPUs) and is capable of delivering real-time frame rates.

CR Categories: H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems—Artificial, augmented, and virtual realities; I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms

Keywords: augmented reality, rendering, photometric registration, photographic imperfections, image noise, motion blur, aliasing

1 Introduction

In augmented reality (AR), graphical objects are overlaid on top of the view of the real world [2]. In video see-through augmented reality, this is achieved by continually acquiring digital images from a camera which captures the real environment. These camera images serve as background in the image mixing process, and the virtual objects are rendered over them. In order to achieve a spatially correct overlay of the virtual scene elements, camera tracking techniques are employed. Among the most popular and easy-to-use tracking methods are vision-based tracking systems like the ARToolKit [16] and ARTag [4].

In most augmented reality systems, standard real-time rendering methods are used in order to generate the graphical representation of virtual objects. Widespread graphics libraries like OpenGL or high-level toolkits and scene graphs based on these libraries are often used for this task. While these standard renderers typically are very fast and easy to integrate in an AR application, they also produce the characteristic artifacts of real-time computer graphics. The generated representation of virtual objects is based on manually defined artifical light sources and material parameters. Simple interpolation methods are used for spreading the computed color values over the area of the polygons which make up the graphical objects contained in the augmented environment. Moreover, artefacts like strong aliasing at the outer boundary of virtual objects make them easily discernible from the camera image.

Altogether, the virtual scene elements in an augmented environment typically have an artifical and almost "plastic-like" look. This leads to significantly different levels of realism of the camera image and the graphical objects, which therefore stand out from the background. This "realism gap" can be considered one of the main challenges in rendering for augmented reality.

One effect contained in the digital camera image, which is not present in the renderings of virtual objects, is image noise. This image noise is generated by the CCD chip of the digital video camera that is used for capturing images of the real environment. (CCD is an acronym for charge-coupled device, which is the technical term for the commonly used type of digital image sensor.) There are different physical reasons for this noise, e.g., photodiode sensitivity variations and photon noise [14]. Depending on the quality of the CCD chip and the environmental conditions, the impact of the noise can vary from being a subtle phenomenon to a clearly discernible variation in the digital video stream. Figure 1 shows an example of camera image noise delivered by a webcam.

In this paper, a novel approach to rendering virtual objects, which mimics the noise contained in the camera video stream, is proposed. A simple calibration step is described, which measures image variations relative to averaged color intensities in a static scene. The characteristics of the camera image noise are then estimated as the mean and standard deviation of a normal distribution of variations in color channel intensities. These characteristics are used in a modified augmented reality rendering pipeline, which simulates noise when displaying virtual objects. This animated noise simulation contributes to equalizing the realism in the camera image and the virtual objects.

A second peculiarity of images delivered by a digital video camera is motion blur. Motion blur results from the temporal integration of color intensities on the CCD chip [19]. If there is fast motion in the captured scene, due to fast movement of the camera or observed objects, color intensities corresponding to different real objects are averaged in the image over time. This leads to a blurred reproduction of the real scene in the camera image. Figure 2 illustrates the motion blur effect in an image acquired from a webcam.

Here, a method for recreating motion blur in the graphical representation of virtual objects is presented. The size and direction of

*e-mail: fischer@gris.uni-tuebingen.de

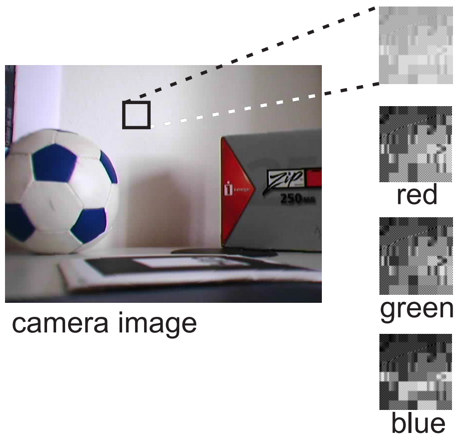


Figure 1: Example of noise in the camera image. An image captured with a webcam is shown. On the right hand side, an enlarged image detail of a homogenuous image region is depicted with enhanced contrast. Contrast-enhanced representations of the three color channels of the same image detail are also shown.

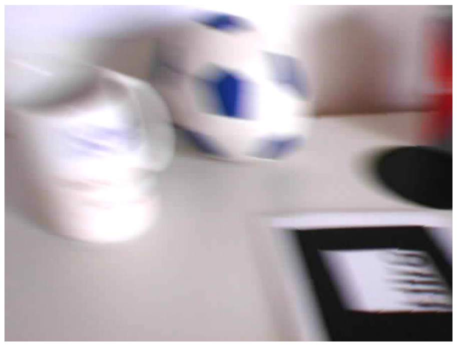


Figure 2: Example of motion blur in the camera image.

the blur is approximated based on the motion of the projected virtual objects in camera image space in subsequent frames. An iterative rendering algorithm then draws overlapping transparent copies of the virtual scene element along the blur vector, resulting in a visual effect similar to motion blur.

The implementation of both the image noise simulation method and the motion blur renderer utilize the programmability of modern graphics processing units (GPUs). The resulting rendering pipeline is capable of generating output images at real-time frame rates.

The adaptation of the visual realism of virtual objects to the realism of the camera image is a major challenge in augmented reality rendering. Relatively little previous work exists in the area of specialized display algorithms for AR, and to date none has dealt with camera image noise and motion blur effects. Therefore, the methods proposed in this paper represent a significant contribution to the ongoing research in AR rendering.

In the remainder of this paper, an overview of related work is given in Section 2. Section 3 discusses the new method for simulating camera image noise. The method for the smooth blending between camera image and virtual object pixels is presented in Section 4. The motion blur rendering technique is described in Section 5. Results obtained the new AR image generation methods are presented in Section 6, and Section 7 concludes the paper with a summary.

2 Related Work

The methods proposed in this paper aim at adapting the visual realism of virtual objects to the appearance of the camera image. They can therefore be considered *photometric registration* techniques. The photometric registration problem is defined as the task of adapting the illumination conditions and overall visual appearance of two images. An early method for the correct automatic illumination of virtual objects added to real images was described by Debevec [3]. Research has also been done into methods of analyzing the real illumination conditions in an interactive augmented reality setup. Examples of this approach include the work of Gibson et al. on photometric reconstruction for mixed reality [11]. The system of Kanbara and Yokoya analyzes the distribution of real light sources, which is then used for adapting the representation of graphical objects [15]. Their method requires a special marker and mirror ball to be visible in the camera image in order to compute the environment light map. A similar technique, which also utilizes an acquired environment illumination map, was proposed by Agusanto et al. [1]. In their system, a mirror ball and special camera are used in a specific preparatory procedure for determining the lighting conditions in the scene. Heymann et al. have described a GPU-accelerated rendering technique for realistic illumination in AR based on information from a captured mirror sphere [13].

An advanced type of photometric registration is the AR rendering method developed by Okumura et al. [17]. Their system also applies blur to the graphical representation of virtual objects. However, this system is not designed to handle motion blur, but deals with the blur caused by real objects being out of the focal plane of the camera.

Recently, a different approach to equalizing the visual realism of real and virtual scene elements was proposed. By applying similar non-photorealistic stylization filters to both the camera image and the computer-generated graphics, *stylized augmented reality* generates a homogeneous visual appearance in the output video stream [5, 8]. Algorithms for a cartoon-like [7] and a painterly stylization [6] of augmented video streams have been presented. It was shown that the application of a stylization filter in augmented reality significantly reduced the discernability of virtual objects in an experimental study [9]. Haller et al. developed a method for displaying both the camera image and virtual objects in AR in a “loose and sketchy” style [12].

The drawback of stylized augmented reality techniques is that they significantly alter the camera image. Most types of artistic or illustrative stylization remove details from the input image by creating large homogeneous regions or composing the output image of relatively large brush strokes. While this is acceptable for some applications, e.g. entertainment and art installations, generating a stylized camera image is not appropriate in other scenarios. The techniques presented in this paper do not have this disadvantage. The virtual objects are adapted to the visual appearance of the camera image, which is preserved in its original, unprocessed form.

3 Image Noise

In this paper, a model for simulating camera image noise is presented. It is assumed that a given digital camera capturing the background image for AR delivers a distinctive type of image noise. The noise characteristics of this image noise are measured using a simple calibration step. These characteristics are described by a simple set of parameters. They are then used in an adapted rendering pipeline for virtual objects in augmented reality, which recreates the measured image noise.

3.1 Image Noise Model

In the literature, precise theoretical descriptions of CCD noise have been discussed (e.g., see Withagen et al. [21]). However, such an exact noise model can only be established based on elaborate measurement procedures under controlled conditions. Moreover, Withagen et al. conclude that the noise produced by a typical commercially available webcam does not conform well to the common exact noise model. Since most video see-through augmented reality systems use commercially available webcams, a simplified noise model, which can be estimated based on an uncomplicated calibration procedure, is used here.

In this simplified noise model, it is assumed that the variation of intensity in each color channel is normally distributed. We denote the difference between the captured color channel value, $c_{\{r,g,b\}}$, and the ideal color channel value in the image of the actual scene, $i_{\{r,g,b\}}$, as channel variation $v_{\{r,g,b\}} = c_{\{r,g,b\}} - i_{\{r,g,b\}}$. This means that the probability density function of the channel variation $v_{\{r,g,b\}}$ can be expressed with the well-known Gaussian function, as shown in Equation 1. In the Gaussian function, the characteristics of the normal distribution are described by the mean, μ_i, and the standard deviation σ_i. The resulting $f(v_i\,;\,\mu_i,\sigma_i)$ is the likelihood of a channel variation of v_i from the ideal image color in a pixel. Note that each color channel, $\{r,g,b\}$, is treated separately, which leads to three different mean values and standard deviations, $\mu_{\{r,g,b\}}$ and $\sigma_{\{r,g,b\}}$.

$$f(v_i\,;\,\mu_i,\sigma_i) = \frac{1}{\sigma_i\sqrt{2\pi}} exp\left(-\frac{(v_i-\mu_i)^2}{2\sigma_i^2}\right), \quad i \in \{r,g,b\} \qquad (1)$$

As a refinement of the basic noise model introduced in Equation 1, different normal distributions are not only established for each color channel, but different sets of distributions are used depending on the overall pixel intensity. Some effects contributing to CDD noise are proportional to the magnitude of the incoming light signal (e.g., shot noise [21]). In the system proposed here, this phenomenon is modelled by estimating several sets of normal distribution parameters, $\mu^j_{\{r,g,b\}}$ and $\sigma^j_{\{r,g,b\}}$. The average intensity of a captured image pixel, $\bar{c}(x,y)$, is computed as shown in Equation 2.

$$\bar{c}(x,y) = \frac{c_r(x,y)+c_g(x,y)+c_b(x,y)}{3} \qquad (2)$$

The determined average intensity is then quantized into one of N intensity bins, i.e., $j = \lfloor \bar{c}/N \rfloor$. In the current implementation of the system, a number of intensity bins equal to a power of two is always used. This means that image intensities are partitioned into $N = 2^k$ bins, each spanning an intensity range of the size $256/2^k = 2^{(8-k)}$ for the typical 8-bit camera images. For each of the intensity bins, a normal distribution is estimated, as shown in Equation 3. This Gaussian function expresses the probability of a channel variation v^j_i if the average intensity of the captured image pixel falls into the intensity bin j.

$$f^j(v^j_i\,;\,\mu^j_i,\sigma^j_i) = \frac{1}{\sigma^j_i\sqrt{2\pi}} exp\left(-\frac{(v^j_i-\mu^j_i)^2}{2(\sigma^j_i)^2}\right)$$
$$i \in \{r,g,b\}, \quad j \in \{0,\ldots,N-1\} \qquad (3)$$

3.2 Image Noise Calibration

An easy-to-use calibration step for determining the noise distribution parameters (μ^j_i, σ^j_i) for all channels i and all intensity bins j is proposed. This calibration step is based on the principle of capturing one or several arbitrary but static scenes with the webcam over a duration of several frames. We refer to these static scenes as *reference scenes*, and the number of reference scenes is denoted as S. For each of the reference scenes, several frames are recorded. The number of these *reference frames*, for a reference scene s, is called $F^s, s \in \{1,\ldots,S\}$.

An interactive software tool was implemented, which allows the user to capture reference scenes with arbitrary durations. This results in a series of stored reference frames r^{f_s} with frame numbers $f_s \in \{1,\ldots,F^s\}$ within the image sequence captured for reference scene s. Subsequently, the noise parameter estimation process is started by the user.

At the beginning of the noise distribution parameter estimation, average images are computed for the reference scenes. For each reference scene, the average image a^s is calculated according to Equation 4. As shown in Equation 4, the pixel intensities $r^{f_s}_i$ are summed up over all frames f_s of a reference scene for each color channel $i \in \{r,g,b\}$. The resulting sum is then devided by the number of reference frames in this scene, F^s. This computation yields the averaged image a^s for reference scene s.

$$a^s_i(x,y) = \frac{1}{F^s}\sum_{f_s=1}^{F^s} r^{f_s}_i(x,y)\;, \;\; i \in \{r,g,b\}, s \in \{1,\ldots,S\} \qquad (4)$$

The average images a_s are assumed to be the “ground truth”, i.e., the ideal pixel values for the actual observed static scene. It is now possible to determine the noise in each pixel of each reference frame relative to its associated average image. The computation of these measured color channel variations $v^{f_s}_i$ is shown in Equation 5.

$$v^{f_s}_i(x,y) = r^{f_s}_i(x,y) - a^s_i(x,y)\;, \;\; i \in \{r,g,b\}, f_s \in \{1,\ldots,F^S\} \qquad (5)$$

As the next step, histograms of the color channel variations are computed. For each color channel and each intensity bin, one histogram is constructed. This is achieved by iteratively processing the color channel variations in all pixels of all reference frames. For a given color channel variation $v^{f_s}_i(x,y)$, the average intensity of the associated average image pixel is calculated, i.e., $\bar{a}^s_i(x,y)$. Subsequently, the corresponding intensity bin $j = \lfloor \bar{a}/N \rfloor$ is determined. The color channel variation is then counted as one occurrence of the varation size $v^{f_s}_i(x,y)$ in the histogram for color channel i and intensity bin j.

The result of the histogram construction process are $3 \cdot N$ histograms, $histogram^j_i(v)$ with $i \in \{r,g,b\}$, $j \in \{0,\ldots,N-1\}$, and $v \in \{-255,\ldots,255\}$. Note that each histogram has 511 entries for measured color channel variations between -255 and 255, which are the maximum possible variations in 8-bit images. The content of each entry represents the number of occurrences of the corresponding color channel variation. An example of a channel variation histogram is shown in Figure 3. In this plot, data measured for a Sony EyeToy webcam are depicted. Only a single intensity bin was used in this example (N=1). As illustrated in Figure 3, the real noise distribution measured in the webcam image data indeed resembles a normal distribution.

Finally, the noise distribution parameters are estimated from the computed channel variation histograms. We use the maximum likelihood estimation for determining the mean values μ^j_i and standard deviations σ^j_i of the measured noise data. (For a description of the maximum likelihood estimation, refer to [20].) The mean value is computed as the sum of all color channel variations divided by the number of pixels, as shown in Equation 6. In this equation, P^j is the total number of pixels over all reference frames which fall into intensity bin j. This number is established by updating corresponding counter variables for each intensity bin during the histograms

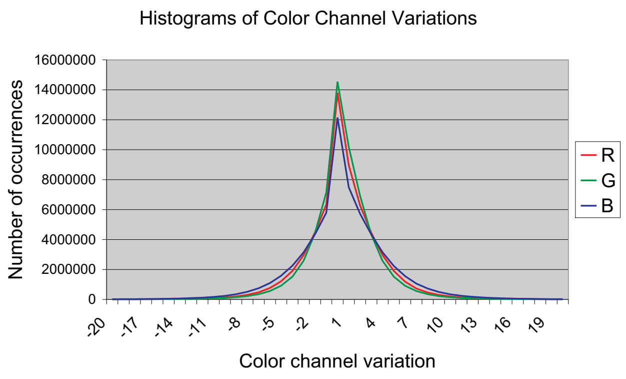


Figure 3: Channel variation histograms for the red, green and blue channels measured for a Sony EyeToy webcam. Only one intensity bin was used here (N=1). The number of reference scenes, S, was 5, and the total number of reference frames was 199. Only a limited portion of the entire channel variation range is shown. Outside of the depicted range, the histogram entries are close to zero or zero.

construction. The summation in Equation 6 is also performed over all pixels in intensity bin j over all reference frames.

$$\mu_i^j = \frac{1}{P^j} \sum_{\substack{\text{all pixels}\\ \text{in bin } j}} v_i^j(x,y) \quad , \; i \in \{r,g,b\} \tag{6}$$

The standard deviations for the channel varations are calculated as shown in Equation 7. Again, a summation over all P^j pixels associated to intensity bin j is performed. However, this time the squared differences between the channel variations and their mean are summed up. The resulting sum is again divided by the total number of relevant pixels, yielding the squared standard deviation $(\sigma_i^j)^2$. (The squared standard deviation is equal to the variance of the distribution.)

$$(\sigma_i^j)^2 = \frac{1}{P^j} \sum_{\substack{\text{all pixels}\\ \text{in bin } j}} (v_i^j(x,y) - \mu_i^j)^2 \quad , \; i \in \{r,g,b\} \tag{7}$$

In the implementation of the noise distribution parameters estimation, optimized methods for the computation of the means and standard deviations are used. These are possible due to the previously computed histogram data. Since the number of occurrences of each channel variation is known, it is not necessary to loop over all pixels in all reference frames again. Instead, the entries of the histograms are iteratively processed. The summation terms are then simply multiplied by the number of occurrences of this channel variation. These expressions replace the corresponding number of summation terms in Equations 6 and 7. The optimized computation of the means and standard deviations is shown in Equation 8.

$$\begin{gathered} \mu_i^j = \frac{1}{P^j} \sum_{v=-255}^{255} histogram_i^j(v) \cdot v \\ (\sigma_i^j)^2 = \frac{1}{P^j} \sum_{v=-255}^{255} histogram_i^j(v) \cdot (v - \mu_i^j)^2 \\ i \in \{r,g,b\}, \quad j \in \{0,\ldots,N-1\} \end{gathered} \tag{8}$$

The final output of the image noise calibration step are the mean values μ_i^j and the standard deviations σ_i^j for all color channels $i \in \{r,g,b\}$ and all intensity bins $j \in \{0,\ldots,N-1\}$. This information is then used for generating corresponding noise textures.

3.3 Generation of Noise Textures

The noise distribution parameters computed in the noise calibration step are stored in a file. They are then read by the actual augmented reality application, which uses an adapted rendering strategy for displaying virtual objects with overlaid noise. In this application, noise textures are generated during initialization as a preparation step for the actual rendering algorithm.

The noise textures are generated as RGB textures, in which each texel holds random channel variation values. These channel variations are generated such that they have a distribution corresponding to the measured noise mean values and standard deviations. In order to be able to store the channel variation values in the texture, they are normalized to be in the 8-bit range of [0;255], corresponding to channel variations between -128 and 127.

A special random number generator has to be used so that random values with a normal distribution of a given mean and standard deviation are generated. This is not possible with the standard C++ random number generator, which produces an uniform distribution (i.e., each value in the random number range has the same probability). In the implementation of the system presented here, the non-uniform random number generators library from [10] is used. This library is capable of producing random numbers with a definable normal distribution.

A user-defined parameter determines the size of the generated noise textures. In the implementation of the system, quadratic textures with a side length equal to a power of two are always used. For each of the intensity bins defined in the calibration step, a noise texture is generated. The texels of each texture are computed in an iterative process which loops over the entire area of the texture. The values of the color channels of the currently regarded texel are obtained from a normally distributed random number generator with distribution parameters (μ_i^j, σ_i^j). An example of a noise texture is depicted in Figure 4.

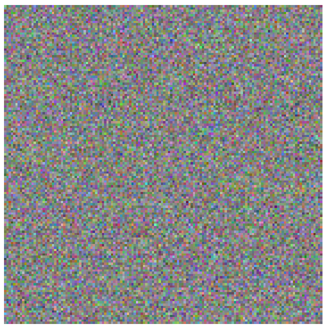


Figure 4: Enlarged portion of a 512×512 noise texture generated with the distribution parameters derived from the histogram shown in Figure 3. A color channel variation of 0 is represented as 50% grey (intensity level 128), negative channel variations are darker, positive variations are brighter. The color channel varations shown here were multiplied with the factor 8 in order to improve the visibility of the noise data.

3.4 Incorporating Noise in the Rendering

After the noise textures have been generated, they are used in an adapted rendering pipeline, which displays the virtual objects with overlaid noise. This rendering system is based on a multipass approach. At first, the steps of a conventional AR pipeline, i.e., the display of the camera image and the drawing of graphical objects, are performed. The image data generated in these steps are read back into texture memory. Then, an image processing step combines the camera image, the rendered virtual objects, and the noise textures into a final output image. Figure 5 shows an overview of this rendering pipeline.

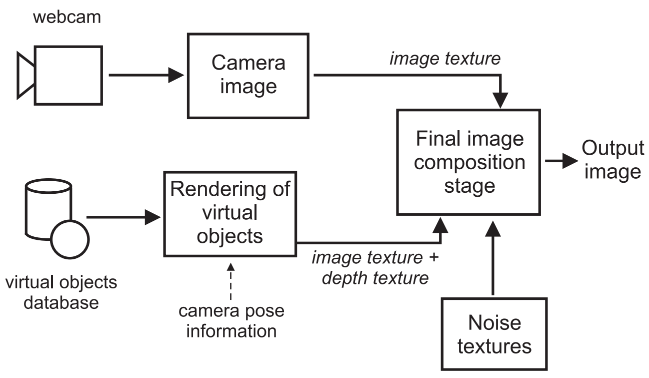


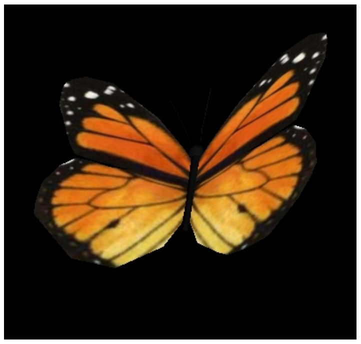
Figure 5: Diagram of the augmented reality rendering pipeline for overlaying noise over the virtual objects.

In the first pass of the adapted AR display pipeline, the camera image is rendered. This is typically done by a special function of the augmented reality framework, which takes internal camera parameters like lens distortion into account. (For example, a specialized camera image display function is provided in the ARToolKit library [16].) After the camera image has been rendered into the currently invisible back frame buffer, it is read back into texture memory. In the implementation of our system, this readback is performed with the OpenGL function `glCopyTexSubImage2D()`, which copies the color values of the rendered camera image into an RGB texture.

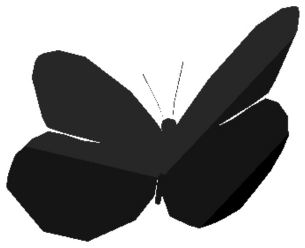
Before the second pass of the rendering pipeline is executed, the back frame buffer is cleared, so that it contains only black pixels. Next, the virtual objects of the augmented environment are rendered. They are transformed according to the current estimated camera pose, which is obtained for instance from the marker tracking component of the AR framework. After the virtual objects have been rendered, the contents of the back frame buffer are again read back into texture memory. This time, however, both the contents of the RGB color buffer and the depth buffer are copied into separate textures. (A specific depth texture format is provided by OpenGL for making the readback of depth buffer contents possible.) Figure 6 depicts the color and depth buffer contents after rendering a virtual butterfly model.

The image data acquired in the first two rendering passes are then combined into the final output image. This step of the display pipeline utilizes the programmability of modern graphics processing units (GPUs). A specialized shader program, which performs the required image processing operations, was developed. It was implemented using the OpenGL Shading Language [18].

For each pixel in the output image, this noise shader looks up the color and depth values at the corresponding location of the image textures which were acquired in the previous rendering passes. It then decides whether a virtual object pixel is to be rendered at the current coordinates. This is achieved by comparing the obtained depth value with the far limit of the depth range, which is stored in the current OpenGL state. If the depth buffer value at a location is less than the value of the depth range limit, a virtual object pixel is present. If this is not the case, the camera image pixel obtained



(a) Frame buffer contents



(b) Depth buffer contents

Figure 6: Rendered representation of a virtual butterfly model. Both the copied frame buffer texture and the depth buffer texture are depicted. The contrast of the depth buffer image was enhanced in order to improve the visibility. In the depth image, darker values correspond to pixels closer to the viewer.

from the first rendering pass is displayed as the output of the final image composition step.

In the implementation of the shader, the comparison of depth values and selection of output pixels is performed with the `step()` and `mix()` functions provided by the OpenGL Shading Language. The use of conditional branching is avoided, as it is a notoriously slow operation on GPUs.

If a virtual object pixel has been detected at the current location, its color values are modified based on the precomputed noise textures. Initially, noise texture coordinates are defined such that the textures are repeated over the entire area of the output image in a straightforward manner. Then, random variations are applied to the texture coordinates in order to create varying, animated noise in the output video stream. Two types of variations are used in the shader. First, random offsets are added to the texture coordinates. This means that the noise textures are translated by a random vector. As the second variation, the texture coordinates are transformed by a random rotation. This can easily be achieved by multiplying the texture coordinates with the corresponding rotation matrix in the shader. For each frame, a new random translation vector and a new random rotation angle are determined. These random values are generated on the CPU by the software which manages the image processing shader, and they are then passed to the shader using shading language functionality. (Currently, existing implementations of the OpenGL Shading Language are not yet capable of producing random values in a shader program.)

The resulting, randomly modified texture coordinates are then used for looking up the channel varations in the noise texture. The average intensity of the original virtual object pixel is computed, and then the noise texture corresponding to the associated intensity bin is selected. Finally, the channel variations stored at the determined texture coordinates in this texture are added to the color of the output pixel.

The result of this process is the final output image. The camera image is displayed as it was originally rendered. The virtual objects are displayed with an overlaid animated noise texture, the characteristics of which correspond to the data measured in the noise calibration step. Figure 7 shows the effect of the noise overlay on the virtual butterfly model.

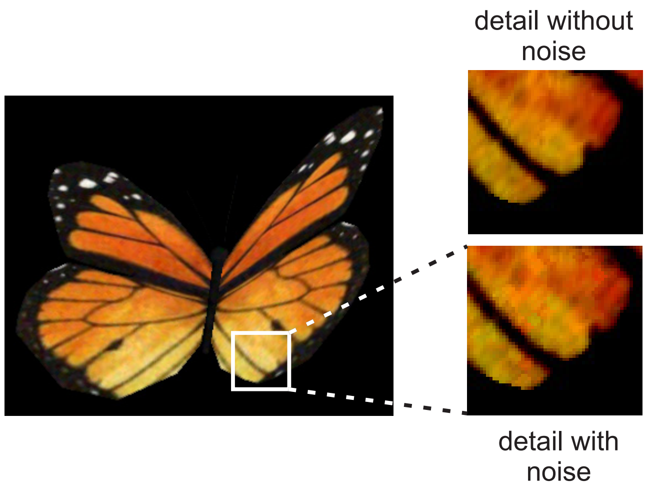


Figure 7: Rendering of the virtual butterfly model with overlaid noise. (The contrast in the enlarged image details was enhanced in order to improve the visibility of the noise.)

3.5 Refinements of the Noise Rendering Algorithm

After initial experiments, several refinements were introduced into the described basic noise rendering pipeline. These were developed based on an empirical assessment of the visual appearance of the noise overlays generated by the display pipeline.

The first improvement allows for the possibility of adapting the size of the noise texels. In the original pipeline, each noise texel modifies exactly one output pixel. This does not correspond well to the color variations in the camera image. In our experience, the color variations in the camera image often affect patches of several pixels. A possible explanation for this effect could be artifacts generated by the camera image rendering process. This rendering process displays a scaled and warped representation of the original camera image data in order to take the internal camera parameters into account. Figure 8, which shows the same image detail as depicted in Figure 7, illustrates the effect of magnifying the noise texels. In this example, a noise texel magnification factor of 4.0 was used, leading to a noise texel size of 4x4 pixels. In the implementation of the system, the noise texel magnifier is applied by dividing the rotated and translated texture coordinates by the magnification factor. Nearest neighbour interpolation is used for the texel lookup, resulting in discrete quadratic patches in the output image which are affected by a single noise texel. (Figure 8 also nicely demonstrates the random rotation of texture coordinates described in the previous section.)

As a second refinement of the original noise rendering pipeline, a noise scaling factor, which can be defined by the user, was added. The color channel variations obtained from the noise textures are

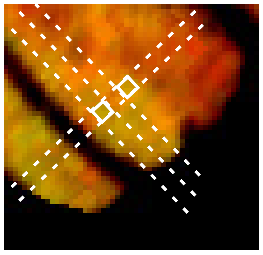


Figure 8: Illustration of the noise texel magnification factor. The dotted white lines represent boundaries between (rotated) columns and rows of noise texels. The solid white boxes highlight two individual noise texels. For this image, a noise texel scaling factor of 4.0 was used.

multiplied with this factor before they are added to the virtual object pixels in the noise shader. This way, the impact of the noise texture can be manually adapted in order to achieve a stronger and more visible effect.

Finally, experiments with the original noise rendering system revealed that the changes in the noise overlay happened too often. This resulted in a strong flickering effect instead of a natural noise-like appearance. Therefore, a frame counter with a user-definable update delay d was introduced. Thanks to this counter, the random variations of the texture coordinates (i.e., random offset and rotation angle) are updated only every d frames. This results in a more slowly varying noise, which corresponds better to the noise in the camera image.

4 Antialiasing

As a by-product of the new noise rendering pipeline, it becomes possible to perform a smooth blending between virtual objects and the camera image. Due to the preceding readback of color and depth buffer contents, all the data required for such an antialiasing step are available.

In order to be able to perform the blending, a 3x3 neighbourhood of color and depth texels is looked up for each output pixel. For each of the texels in the neighbourhood it is then decided whether a virtual object or the camera image is visible at that location. As described in Section 3.4, this can be determined with a simple comparison of the obtained depth value with the far depth range limit. The computation of this decision variable $D(x,y)$ at the location (x,y) is shown in Equation 9. In this equation, the resulting decision variable is assigned a value of one if the depth value at this location, $depth(x,y)$, is less than the current far depth range limit, $depthRange_{far}$. In this case, a virtual object pixel has been detected. Otherwise, a value of zero is assigned to $D(x,y)$, which indicates that a camera pixel is detected there.

$$D(x,y) = \begin{cases} 1, & depth(x,y) < depthRange_{far} \\ 0, & otherwise \end{cases} \tag{9}$$

If a camera image pixel was detected at the currently regarded location, i.e., $D(x,y) = 0$, the color value of the corresponding camera image texel is used as output. Otherwise, the result of the blending operation is used. This blending is computed as shown in Equation 10. The blending operation is designed so that an averaging of color values is only done for boundary pixels, i.e., those virtual object pixels which have adjacent camera pixels. For such a virtual object boundary pixel, only the neighbouring camera image pixels are taken into account for the blending, while adjacent virtual object pixels are ignored.

Therefore, a sum of camera image pixels $c(x+i, y+j)$ is computed over the 3x3 neighbourhood as in Equation 10. Only those pixels, for which a depth decision variable $D(x+i, y+j)$ of zero has been determined (i.e., camera pixels), are taken into account. The color of the central virtual object pixel, $v(x,y)$, is also added to the result of this summation. The resulting summed up color data is then divided by factor n, which is equal to the number pixels which where included in the summation (i.e., the number of adjacent camera image pixels plus one). The final result is the antialiased virtual object boundary pixel color $b(x,y)$.

$$b(x,y) = \frac{1}{n}\left(v(x,y) + \sum_{i=-1}^{1}\sum_{j=-1}^{1}(1-D(x+i,y+j))\cdot c(x+i,y+j)\right) \tag{10}$$

Due to this averaging of color values between virtual object boundary pixels and adjacent camera image pixels, a smooth blending is performed at the object boundaries. This effect is illustrated in Figure 9.

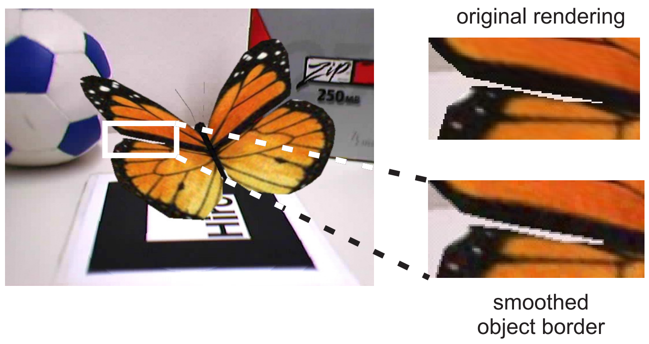


Figure 9: Smooth blending at the boundary of the virtual butterfly model. As shown on the right hand side, the aliasing between the camera image and the virtual object is reduced significantly by averaging the color values. (In this example, a smoothing factor $\beta = 0.8$ was used.)

As a refinement of the described blending procedure, a smoothing factor $\beta \in [0;1]$ was introduced. This smoothing factor is used for mixing the original virtual object color into the output pixel with a percentage of $(1-\beta)$. In this way, the impact of the smoothing effect can be adapted by the user.

5 Motion Blur

The second camera image effect which is treated in this paper is motion blur. As explained in Section 1, motion blur results from the temporal integration of pixel intensities. If there is fast movement in the observed scene, colors corresponding to different real objects are averaged, leading to a blurred camera image. Such fast movements in the image can be caused by both changes of the camera pose and moving objects in the scene.

In addition to the noise renderer described in Section 3, we therefore propose a second new display technique for virtual objects in AR, which takes motion blur into account. In order to be able to mimic the effects of motion blur in the camera image, the magnitude and direction of the blurring have to be known. Our system uses an uncomplicated method for approximating this motion blur vector based on the available camera pose information.

The current implementation of our method can render motion blur for a single virtual object. It could, however, be easily extended for handling several objects. In a preprocessing step, the geometric center of the virtual model is computed. This is achieved by finding the bounding box of the model, i.e., the maximum and minimum occurring vertex coordinates. The geometric center of the model is then determined by calculating the center between the maximum and minimum coordinates for each coordinate axis.

During the runtime of the augmented reality system, a blur vector is estimated in every frame. This approximated blur vector is defined as the two-dimensional motion of the center of the virtual object in image space. The computation of this vector is shown in Equation 11. In this equation, the 2D position of the virtual object at time t is denoted as $pos(t)$. This position is determined by projecting the geometric center of object, *center*, into image space. The projection operation, expressed as $proj_{Pose(t)}$, is performed according to the currently estimated camera pose, $Pose(t)$. This camera pose is the same as that used for rendering the virtual objects in the AR scene.

The projected 2D position of the object center is calculated in every frame, and the last determined position is stored. It is therefore possible to compute the difference between the last position, $pos(t-1)$, and the current position, $pos(t)$. This difference vector represents the 2D motion of the virtual object in image space and is used as an approximation for the blur vector.

$$\begin{aligned} pos(t) &= proj_{Pose(t)}(center) \\ blurVector(t) &= pos(t) - pos(t-1) \end{aligned} \tag{11}$$

5.1 Rendering Motion Blur

In every frame, the rendering system decides whether to display the virtual object with or without motion blur. This is done by comparing the length of the blur vector with two thresholds. The first threshold defines the minimum object movement required to make the use of motion blur useful (typically between 5 and 10 pixels). The second threshold helps to ignore blur vectors resulting from an erratic pose estimation, e.g., if no useful pose data was delivered by the marker tracking component due to an occluded marker. This second threshold is typically rather large, well beyond 100 pixels. The motion blur technique is applied only if the length of the blur vector is between these two thresholds.

If motion blur rendering is to be applied in a time step, an adapted display method is used. This display method is based on separately generated image and depth textures containing the virtual object, similar to the rendering pipeline described in Section 3.4. The virtual object is rendered in a separate step according to the current camera pose, and the resulting color and depth buffer contents are copied into texture memory. Subsequently, the camera image is rendered, overwriting the contents of the frame buffer. The camera image serves as background for the motion blur rendering technique.

The simulated motion blur effect is created by repeatedly blending the virtual object texture over the camera image at different positions. These positions are obtained by adding offsets to the 2D position of the object, $pos(t)$. We assume that the computed blur vector represents the magnitude and direction of the *current* object motion. (We make this assumption despite the fact that the blur vector is estimated based on retrospective data. This is, however, only a minor simplification and has proven to deliver a good approximation of the motion blur.) Object rendering positions are generated along a line which is centered at the current object position and has the same length and direction as the blur vector.

The first center position for blending the virtual object texture over the camera image is located at $pos(t) - 0.5 \cdot blurVector(t)$, i.e., at the beginning of the line. Then, new center positions are iteratively computed by adding a vector with the same direction as the blur vector to the current position. This vector has a length of $stepSize$, which is a user-definable variable determining the distance between two copies of the virtual object. Therefore, the current center position is modified by the addition of the vector

$stepSize \cdot \frac{blurVector(t)}{|blurVector(t)|}$ in each iteration. This loop is iterated until the end of the line at $pos(t) + 0.5 \cdot blurVector(t)$ is reached. We typically use a step size of close to one pixel for the generation of the center positions. An illustration of virtual object blending center positions for an example blur vector is shown in Figure 10.

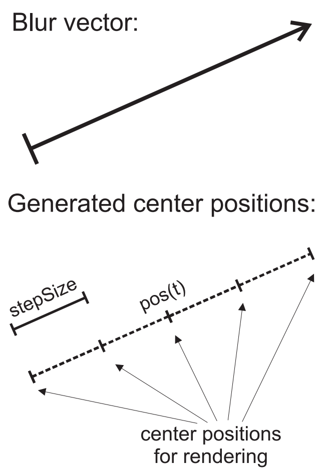


Figure 10: During the motion blur rendering process, multiple copies of the virtual object are rendered. These copies are centered at positions along a line with the same length and direction as the blur vector. This line is centered at the current projected 2D object position. The center positions are generated with a definable step size.

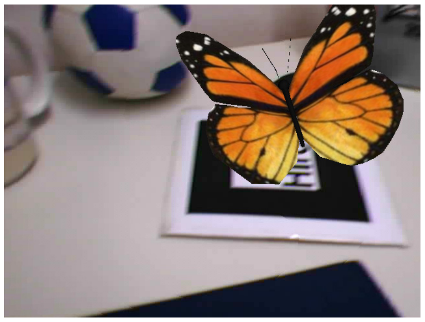
At each of the generated center positions, a copy of the virtual object is drawn. A special shader program is used, which uses the previously rendered virtual object texture as input data. This shader program performs the same depth test as described in Section 3.4 and 4, and it only displays virtual object pixels. This is necessary so that the black background from the virtual object rendering pass is ignored in the display process. Each copy of the virtual object is drawn semitransparently over the image using OpenGL blending. The transparency used for each copy, expressed by the alpha value, depends on the number of copies. It is calculated as shown in Equation 12. In this equation, at first the number of virtual object copies, *numSteps*, is calculated. The alpha value is then computed as 1 divided by *numSteps*.

$$numSteps = \left\lfloor \frac{|blurVector(t)|}{stepSize} \right\rfloor \qquad \alpha = \frac{1}{numSteps} \tag{12}$$

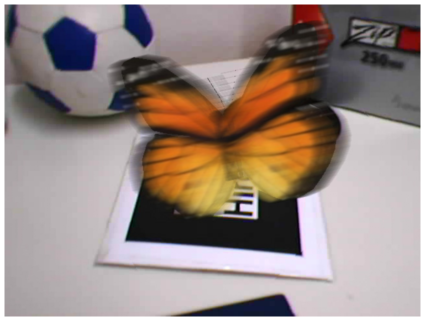
In addition to the copies of the virtual objects spread along the blur vector, another copy is rendered at the original center location. This copy is displayed with a much greater opacity (i.e., a large alpha value). It prevents the camera image from shining through the blurred virtual object. An example of the simulated motion blur effect is shown in Figure 11.

6 Results

We have tested the noise calibration method with several different webcams. Here, we present noise distribution parameters for two models. The first is a Sony Eye Toy USB webcam. The second is a Unibrain Fire-i camera, which uses the Firewire interface for transferring the video stream. In these experiments, four intensity



(a) Without motion blur



(b) With motion blur

Figure 11: Example of the motion blur effect demonstrated with the virtual butterfly model. Note that the object is displayed without blur by the conventional renderer (Fig. 11(a)), although motion blur is clearly visible in the camera image.

bins were used (N=4). Figure 12 contains plots of the standard deviations of the noise distributions for the Sony Eye Toy camera. As shown in the figure, the standard deviations are different for each color channel. While the standard deviations of the noise distributions are smaller in the case of the green channel, they are larger for the red and blue channels. The standard deviations also depend on the intensity bin, with a peak at the second intensity bin. Figure 13 depicts the estimated standard deviations of the noise distributions for the blue color channel of the Unibrain Fire-i camera. It illustrates that the characteristics of the noise delivered by this camera are different from the Sony Eye Toy webcam. The mean values of the distributions have are not depicted in these diagrams. They are almost zero, as it is to be expected from the differences between image intensities in a video sequence and the corresponding average image.

The estimation of the distribution parameters in the noise calibration step typically takes between several seconds and some minutes of computation time, depending on the number and resolution of the reference frames. The parameter estimation, however, is a preprocessing step, and does not have a negative impact on the real-time performance of the AR system. The estimated noise distribution parameters are persistently stored in a file and are loaded during the initialization of the AR application. The noise calibration results shown in Figure 12 and 13 were obtained from more than 500 reference frames at a resolution of 640x480 pixels for each camera.

The new rendering methods for AR were tested with several virtual models, one of them being the virtual butterfly used in the preceding sections. Another model used in the experiments was the virtual model of a hamburger. Figure 14 shows the virtual hamburger model and the effect of the simulated noise overlay. Bench-

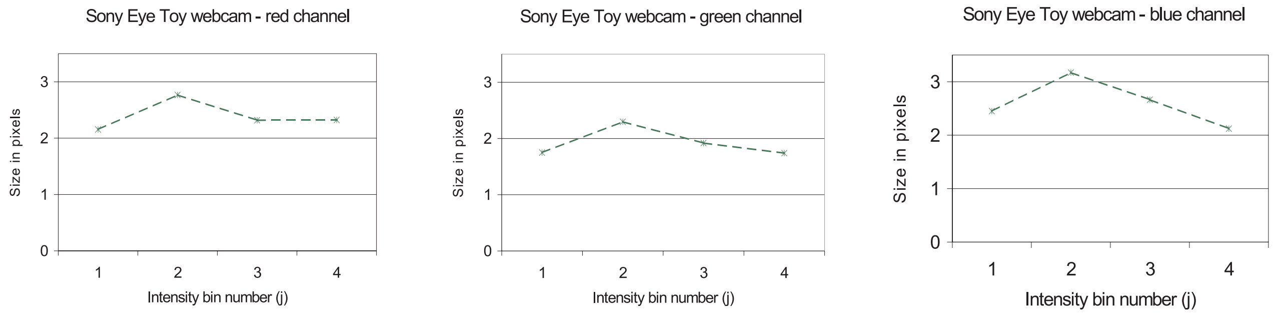


Figure 12: Standard deviations of the noise distributions for the Sony Eye Toy webcam.

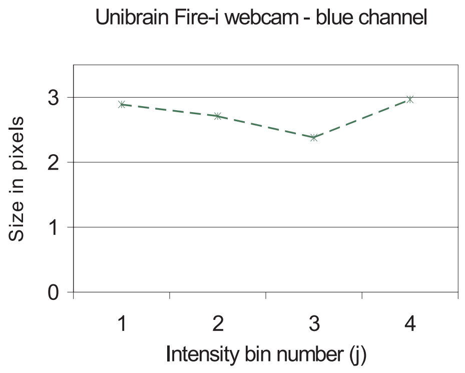


Figure 13: Standard deviations of the noise distributions for the blue color channel of the Unibrain Fire-i webcam.

mark measurements have shown that the noise overlay method has only a very small impact on the frame rate. With conventional AR rendering, frame rates of close to 30 fps were measured (an average of 29.78 fps over 510 frames). With noise rendering turned on, the measured frame rate only dropped to 29.33 fps (averaged over 500 frames). The benchmarks described in this section were performed on a computer with a Pentium 4 processor running at 2.8 GHz using a graphics card with an NVidia GeForce FX 6600 GT chipset. The Unibrain Fire-i webcam, which delives images at a resolution of 640x480 pixels at a rate of 30 Hz, was used for the measurements.

Figure 15 illustrates the smooth blending of virtual object border pixels in the case of the hamburger model. In our implementation of the system, the noise overlay and antialiasing techniques have been combined into a single image processing shader. Therefore, the aforementioned benchmarks for the noise shader include the computation time required for the antialiasing method.

The effect of the motion blur rendering method on the virtual hamburger is depticed in Figure 16. Note that the simulated motion blur rendered for the virtual model corresponds very well to the real motion blur in the camera image. The motion blur technique has a measurable impact on the rendering performance, but is still capable of delivering real-time frame rates. The performance of the motion blur rendering technique strongly depends on the of the length of the blur vector, which determines the number of virtual object copies to be drawn. We measured an average frame rate of 24.36 fps for typical fast motions over a sequence of 520 frames.

7 Conclusions

In this paper, we have presented techniques for mimicking two types of camera image imperfection; image noise and motion blur. Moreover, a method for the smooth blending between virtual objects and the background was described. These new AR rendering

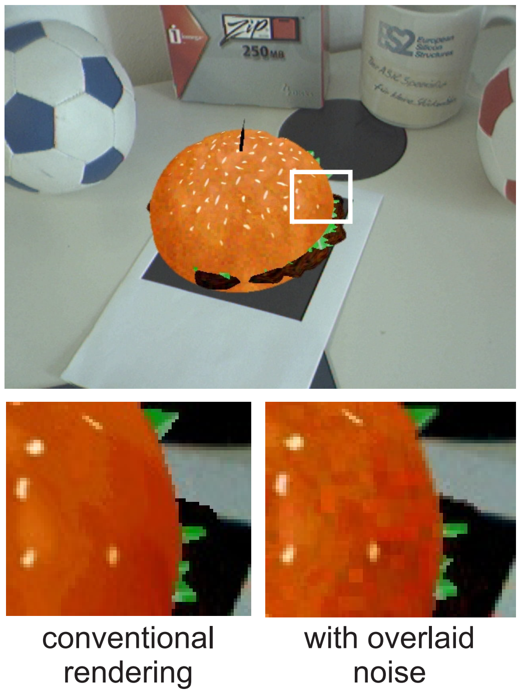


Figure 14: Virtual hamburger model in an augmented environment rendered with overlaid noise. The image details at the bottom compare the conventional rendering of the hamburger and the version with overlaid noise. (The contrast of the details was enhanced to improve visibility.)

techniques have been implemented on the GPU and achieve real-time frame rates.

A simple model for describing camera image noise was introduced. Several simplifications of the full theoretical noise model were used, including the assumption that noise in the individual color channels can be described independently. The presented calibration method does not try to acquire a geometric description of the noise (i.e., the size and shape of the color variations). Instead, the noise texel magnifier creates larger patches of identical color varation. The presented noise model has proven to be a useful and easy-to-use description of image noise. The noise renderer creates a subtle, but perceivable, effect in the output video stream, which helps to equalize the level of realism in the virtual object and camera image.

The motion blur display method uses an approximation of the blur vector, which is based on changes of the estimated camera pose. Since this camera pose estimation is evaluated, the motion

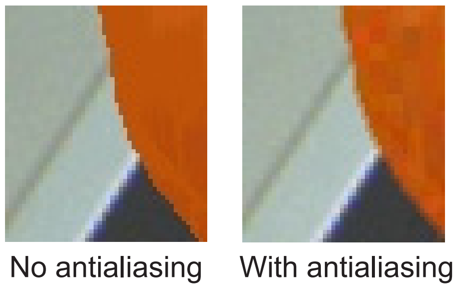


Figure 15: The effect of the smooth blending between camera image and virtual object illustrated for a detail of the hamburger model.

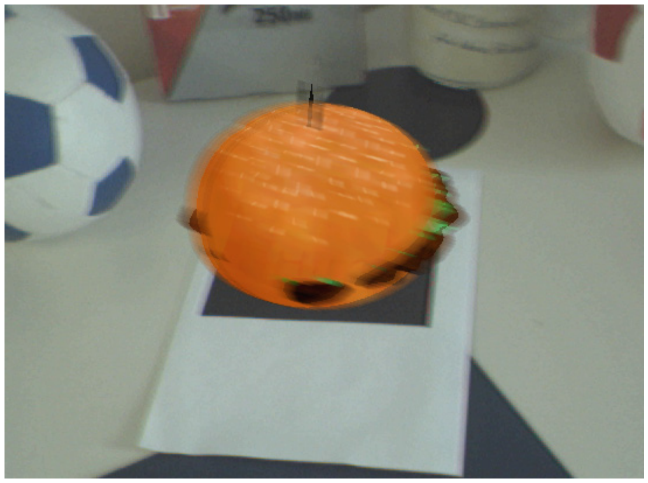


Figure 16: Motion blur applied to the hamburger model.

blur technique can be affected by the tracking lag of the system. Moreover, our current implementation of the motion blur renderer only takes the projected two-dimensional motion of the virtual object into account. Motion blur induced by a quickly changing distance between the camera and the virtual object position is not simulated. Such an extension, however, could be added to the system rather easily. The presented motion blur display algorithm generates a good approximation of the motion blur effect under most circumstances. This technique, therefore, also contributes to making the camera image and virtual object look more similar.

The problem of specialized rendering methods is one of the main challenges in augmented reality. In this area, the task of equalizing the visual realism in the camera image and the virtual objects is very important. Nonetheless, the topic of adapted rendering algorithms for AR has received relatively little attention in the past compared to other main research directions. With this paper, we hope to contribute to the ongoing research in this area and inspire further investigations into the problem.

Acknowledgments

We would like to thank Bernhard Thomaszewski for his suggestions regarding the motion blur problem. We would also like to thank Marta Kersten for proofreading the paper. The graphical models used in our experiments were downloaded from the 3D Cafe website (`www.3dcafe.com`). This work has been supported by project VIRTUE in the focus program on "Medical Navigation and Robotics" (SPP 1124) of the German Research Foundation (DFG).

References

[1] K. Agusanto, L. Li, Z. Chuangui, and N.W. Sing. Photorealistic Rendering for Augmented Reality using Environment Illumination. In *Proc. of IEEE and ACM International Symposium on Mixed and Augmented Reality*, pages 208–216, October 2003.

[2] R. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, and B. MacIntyre. Recent Advances in Augmented Reality. *IEEE Computer Graphics and Applications*, 21(6):34–47, November/December 2001.

[3] P. Debevec. Rendering Synthetic Objects Into Real Scenes: Bridging Traditional and Image-Based Graphics With Global Illumination and High Dynamic Range Photography. In *Proc. of ACM SIGGRAPH*, pages 189–198, July 1998.

[4] M. Fiala. ARTag Revision 1, A Fiducial Marker System Using Digital Techniques. Technical Report NRC/ERB-1117, National Research Council Canada (NRC-IIT), November 2004.

[5] J. Fischer. *Rendering Methods for Augmented Reality*. Ph.D. thesis, Universität Tübingen, 2006.

[6] J. Fischer, D. Bartz, and W. Straßer. Artistic Reality: Fast Brush Stroke Stylization for Augmented Reality. In *Proc. of ACM Symposium on Virtual Reality Software and Technology*, pages 155–158, November 2005.

[7] J. Fischer, D. Bartz, and W. Straßer. Reality Tooning: Fast Non-Photorealism for Augmented Video Streams. In *IEEE and ACM International Symposium on Mixed and Augmented Reality Poster Proceedings*, pages 186–187, October 2005.

[8] J. Fischer, D. Bartz, and W. Straßer. Stylized Augmented Reality for Improved Immersion. In *Proc. of IEEE Virtual Reality*, pages 195–202, March 2005.

[9] J. Fischer, D. Cunningham, D. Bartz, C. Wallraven, H. Bülthoff, and W. Straßer. Measuring the Discernability of Virtual Objects in Conventional and Stylized Augmented Reality. In *Proc. of Eurographics Symposium on Virtual Environments*, pages 53–61, May 2006.

[10] A. Fog. Non-uniform Random Number Generators. http://www.agner.org/random/stocc.htm, 2005.

[11] S. Gibson, T. Howard, and R.J. Hubbold. Flexible Image-Based Photometric Reconstruction using Virtual Light Sources. In *Proc. of Eurographics*, pages 203–214, September 2001.

[12] M. Haller, F. Landerl, and M. Billinghurst. A Loose and Sketchy Approach in a Mediated Reality Environment. In *Proc. of ACM International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia (Graphite)*, pages 371–379, December 2005.

[13] S. Heymann, A. Smolic, K. Müller, and B. Fröhlich. Illumination Reconstruction from Real-Time Video For Interactive Augmented Reality. In *Proc. of International Workshop on Image Analysis for Multimedia Interactive Services*, April 2005.

[14] Isle of Thorns Observatory. CCD's and Noise. http://www.sussex.ac.uk/Units/physics/iotweb/pages/ccds.html.

[15] M. Kanbara and N. Yokoya. Geometric and Photometric Registration for Real-Time Augmented Reality. In *IEEE and ACM International Symposium on Mixed and Augmented Reality Poster Proceedings*, pages 279–280, September 2002.

[16] H. Kato and M. Billinghurst. Marker Tracking and HMD Calibration for a video-based Augmented Reality Conferencing System. In *Proc. of IEEE and ACM International Workshop on Augmented Reality*, pages 85–94, October 1999.

[17] B. Okumura, M. Kanbara, and N. Yokoya. Image Composition Based on Blur Estimation from Captured Image for Augmented Reality. In *Proc. of IEEE Virtual Reality*, pages 128–134, 2006.

[18] R.J. Rost. *OpenGL Shading Language*. Addison-Wesley Publishing Company, 2004.

[19] E. Shechtman, Y. Caspi, and M. Irani. Space-Time Super-Resolution. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(4):531–545, April 2005.

[20] A. Storkey. Learning from Data: Density Estimation - Gaussian Distribution. http://www.anc.ed.ac.uk/˜amos/lfd/lectures/gaussian-print.pdf, October 2005. School of Informatics, University of Edinburgh.

[21] P.J. Withagen, F.C.A. Groen, and K. Schutte. CCD Characterization for a Range of Color Cameras. In *Proc. of Instrumentation and Measurement Technology Conference*, May 2005.