Eurographics Symposium on Rendering (2005)
Kavita Bala, Philip Dutré (Editors)

Motion Blur for Textures by Means of Anisotropic Filtering

J. Loviscach

Hochschule Bremen, Bremen, Germany

Abstract

The anisotropic filtering offered by current graphics hardware can be employed to apply motion blur to textures. The solution proposed here uses a standard texture together with a vertex and a pixel shader acting on a mesh with augmented vertex data. Our method generalizes the usual spatial anisotropic MIP mapping to also include temporal effects. It automatically processes any time series of affine 3D transformations of an object. The application fields include animations containing 2D lettering as well as objects such as spoke wheels that are cookie-cut from large polygons using an alpha channel. We present two different implementations of the technique.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Animation

1. Introduction

Motion blur often affects less the shape of 3D objects than rather their texture. This is true for instance for the ground in driving or flight simulator software, for rotating wheels and globes, and for "flying" 2D logos. On top of that, texture-based motion blur can be applied to flat geometry cookie-cut from larger polygons using a texture with an alpha channel. This applies to air-screws, spoke wheels, and even to sword blades. Another application of texture motion blur is to suppress crawling pixel staircases that occur when a texture with hard edges moves over the screen.

The contribution of this paper is a method to employ standard anisotropic filtering hardware to subject textures to motion blur in real time by vertex and pixel shaders, see Figures 1, 2 and 3. We make use of the `tex2D(s, t, ddx, ddy)` function of HLSL, which allows to specify two vectors `ddx`, `ddy` that span a parallelogram in *uv* space to be used for texture averaging.

This novel use of anisotropic filtering is combined with its original use for spatial antialiasing. As opposed to temporal supersampling, the proposed method does not employ multiple rendering. Thus, there are no issues with transparency (such as rendering order) and depth buffering. In addition, no further buffers are needed, in particular none with extended range such as an accumulation buffer. The method leverages today's specialized hardware to retrieve up to 16 trilinear texture samples efficiently.

Figure 1: *Motion blur is applied to the texture of a wheel rolling on the ground at different speeds. The alpha channel of the texture cookie-cuts the wheel from a quadrangle.*

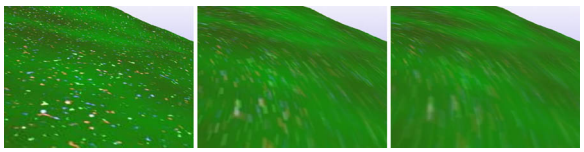

Figure 2: *The presented method prevents "strobe" artifacts in situations such as this flight over uneven terrain (presented at three different speeds).*

This paper is structured as follows: Section 2 gives an overview over related work. In Section 3 we derive how motion blur can be expressed in texture coordinates. How one can combine motion blur with spatial level of detail and the corresponding anisotropy is described in Section 4. Section 5 details our implementations: The first one is heavily pixel-based. It can work with coarsely tessellated geometry.

The second is mostly vertex-based. The third is a fast implementation of multiple rendering, to be used as reference. Section 6 presents and discusses the results. Section 7 summarizes the paper and outlines future work.

Figure 3: *Tumbling billboards and other typographic animations profit from motion blur of the textures. The geometry to which the textures are applied extends beyond the billboards, which are formed by the texture's alpha channel.*

2. Related work

Historically, motion blur—which can also be regarded as temporal antialiasing—has been generated in a variety of ways, including multiple rendering (temporal supersampling) [KB83], integration along motion paths, possibly in Fourier space [PC83], and distribution ray tracing [CPC84]. Recently, Sung et al. [SPW02] have proposed a framework for spatial-temporal antialiasing in an offline renderer using an adaptive Monte-Carlo approach that treats visibility and shading separately.

A standard approach to rendering motion blur at interactive speed is multiple rendering into an accumulation buffer [HA90]. Floating-point buffers offered by some current graphics cards can be employed in the same fashion [Nvi04]. Jones and Keyser [JK04] use a partially CPU-based method to construct polygonal models that describe the volumes the objects sweep in their motion.

Several authors address real-time motion blur by extruding the objects in the direction of their motion and adding an alpha gradient. The basic idea has been proposed by Wloka and Zeleznik [WZ96]. Arce et al. [AW02] present a GPU-based implementation that also addresses sort-order issues of transparent rendering by using six ordered index lists for the triangles. Textures are not treated; furthermore, the algorithm has to render a sharp image in addition. This is more appropriate for still images.

Recent work introduces texture blurring to this deformation method: Tatarchuk et al. [TBI03] use MIP map bias with no precise computation of strength and anisotropy. Green [Gre03] improves on the basic idea by adding motion blur to the textures through temporal multisampling.

To create motion blur for the ground texture of a driving simulation, Hargreaves [Har04] computes different versions of a texture offline, all blurred in different amounts and possibly along different directions. On runtime, a pixel shader combines them according to the current global motion. To cover an arbitrary motion in good quality would require a huge set of precomputed blurred versions of the texture.

Anisotropic filtering is a basic problem in the rendering of textures. A classic approach is elliptical weighted averaging [GH86], where the footprint of a pixel in texture space is approximated by an ellipse. Many authors have proposed more efficient methods to form anisotropic averages, among them Feline [MPFJ99] and SPAF [SLK01].

Current graphics chips assemble an anisotropic footprint by up to 16 trilinear texture samples per pixel, corresponding to a blend of up to 128 texels at different MIP map levels. The manufacturers employ non-disclosed optimization strategies to limit the actual number of requests adaptively. Such strategies are for instance part of ATI Smoothvision HD [ATI04] and Nvidia Intellisample [Nvi03].

3. Motion blur in texture coordinates

Given a triangle moving in 3D space and given a single screen pixel, we want to study the time dependence of the *uv* texture coordinates that correspond to the pixel, see Figure 4. Assume that the triangle has the vertices $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c} \in \mathbb{R}^3$ with *uv* coordinates $\begin{pmatrix} u_a \\ v_a \end{pmatrix}$, $\begin{pmatrix} u_b \\ v_b \end{pmatrix}$, and $\begin{pmatrix} u_c \\ v_c \end{pmatrix}$, respectively. The normalized vector

$$\mathbf{n} := ((\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a}))^0 \qquad (1)$$

is perpendicular to the triangle.

The triangle is subjected to an affine mapping $\mathbf{p} \mapsto M\mathbf{p}+\mathbf{v}$ where M is a 3×3 matrix and $\mathbf{v}$ is translation vector. From one instance of time to the next, M and $\mathbf{v}$ change by small amounts ΔM and $\Delta\mathbf{v}$. We assume that these quantities are small and ignore terms of quadratic or higher order.

Let a fixed screen position correspond to a point $\mathbf{p}$ of the triangle under $\mathbf{p} \mapsto M\mathbf{p}+\mathbf{v}$. After perturbing the mapping by ΔM and $\Delta\mathbf{v}$, a possibly different point $\mathbf{p}+\Delta\mathbf{p}$ is mapped to the same position on screen.

We assume that the mapping from view space to screen space is a perspective projection with the origin as center. This means that $M\mathbf{p}+\mathbf{v}$ and $(M+\Delta M)(\mathbf{p}+\Delta\mathbf{p})+\mathbf{v}+\Delta\mathbf{v}$ lie on the same ray through the origin. Taking differentiability into account, we see that there exists a small Δd such that

$$M\mathbf{p}+\mathbf{v} = (1+\Delta d)\Big((M+\Delta M)(\mathbf{p}+\Delta\mathbf{p})+\mathbf{v}+\Delta\mathbf{v}\Big). \qquad (2)$$

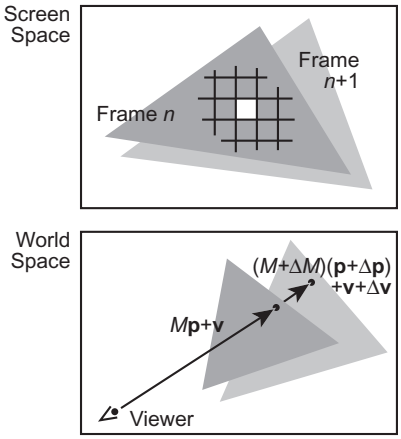

Figure 4: *Let the point* $\mathbf{p}$ *in object space be at the center of a screen pixel in a given frame. Then in the next frame a possibly different point* $\mathbf{p}+\Delta\mathbf{p}$ *will be mapped to the same screen position.*

In general, M is invertible, so that by deleting higher orders we find:

$$\Delta\mathbf{p} = -M^{-1}\Big(\Delta M\mathbf{p} + \Delta\mathbf{v} + \Delta d(M\mathbf{p}+\mathbf{v})\Big). \tag{3}$$

To determine Δd, we can observe that $\Delta\mathbf{p}$ connects two points of the triangle and thus must be perpendicular to $\mathbf{n}$. Therefore, Equation 3 results in

$$\Delta d = -\frac{\mathbf{n}\cdot M^{-1}(\Delta M\mathbf{p}+\Delta\mathbf{v})}{\mathbf{n}\cdot M^{-1}(M\mathbf{p}+\mathbf{v})}. \tag{4}$$

Inserted into Equation 3, this allows to find $\Delta\mathbf{p}$.

The final step is to convert $\Delta\mathbf{p}$ into the corresponding change $\begin{pmatrix}\Delta u\\ \Delta v\end{pmatrix}$ of texture coordinates. Given a point $\mathbf{p}$ on the triangle, we can compute its uv coordinates as

$$\begin{pmatrix}u\\ v\end{pmatrix} = \begin{pmatrix}u_a\\ v_a\end{pmatrix} + \frac{(\mathbf{p}-\mathbf{a})\cdot(\mathbf{n}\times(\mathbf{c}-\mathbf{a}))}{(\mathbf{b}-\mathbf{a})\cdot(\mathbf{n}\times(\mathbf{c}-\mathbf{a}))}\begin{pmatrix}u_b-u_a\\ v_b-v_a\end{pmatrix} + \frac{(\mathbf{p}-\mathbf{a})\cdot(\mathbf{n}\times(\mathbf{b}-\mathbf{a}))}{(\mathbf{c}-\mathbf{a})\cdot(\mathbf{n}\times(\mathbf{b}-\mathbf{a}))}\begin{pmatrix}u_c-u_a\\ v_c-v_a\end{pmatrix}.$$

This formula can easily be verified by inserting $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ for $\mathbf{p}$ and noting that it is linear in $\mathbf{p}$. Taking differences, we finally find $\begin{pmatrix}\Delta u\\ \Delta v\end{pmatrix} = U\Delta\mathbf{p}$, where U is the 2×3 matrix defined by

$$U := \begin{pmatrix}u_b-u_a\\ v_b-v_a\end{pmatrix}\otimes\left(\frac{\mathbf{n}\times(\mathbf{c}-\mathbf{a})}{(\mathbf{b}-\mathbf{a})\cdot(\mathbf{n}\times(\mathbf{c}-\mathbf{a}))}\right)^{\mathrm{T}} + \begin{pmatrix}u_c-u_a\\ v_c-v_a\end{pmatrix}\otimes\left(\frac{\mathbf{n}\times(\mathbf{b}-\mathbf{a})}{(\mathbf{c}-\mathbf{a})\cdot(\mathbf{n}\times(\mathbf{b}-\mathbf{a}))}\right)^{\mathrm{T}}. \tag{5}$$

4. Combining spatial and temporal anisotropy

Standard anisotropic MIP mapping determines the partial derivatives of the texture coordinates u and v with respect to the screen coordinates x and y. These derivatives are used to control which part of the texture map is averaged to compute the color of a pixel at (x,y), see Figure 5.

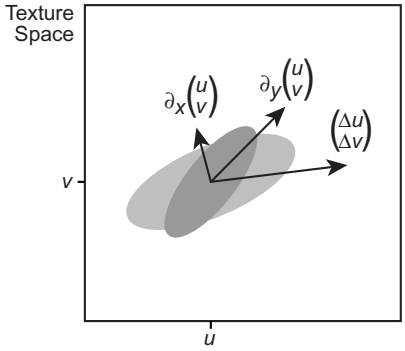

Figure 5: *Spatial anisotropic filtering defines the area of the texture mapped to a single screen pixel (dark gray) using spatial derivatives. For additional temporal filtering (light gray) this footprint has to extend along the motion, too.*

The temporal adjustments have to be combined with these computations. To this end, we use a mixture of Gaussian random variables. For a given pixel at $\begin{pmatrix}u\\ v\end{pmatrix}$, we want to average over the texture values at

$$\begin{pmatrix}u\\ v\end{pmatrix} + \alpha\partial_x\begin{pmatrix}u\\ v\end{pmatrix} + \beta\partial_y\begin{pmatrix}u\\ v\end{pmatrix} + \gamma\begin{pmatrix}\Delta u\\ \Delta v\end{pmatrix}, \tag{6}$$

where α, β, γ are independent identically distributed Gaussian random variables with zero mean and standard deviation σ. We want to use the `tex2D(s, t, ddx, ddy)` function of HLSL. Thus, we seek `ddx` $=: \begin{pmatrix}e\\ f\end{pmatrix}$ and `ddy` $=: \begin{pmatrix}g\\ h\end{pmatrix}$ such that the expression $\begin{pmatrix}u\\ v\end{pmatrix} + \alpha\begin{pmatrix}e\\ f\end{pmatrix} + \beta\begin{pmatrix}g\\ h\end{pmatrix}$ possesses the same probability distribution as does expression 6.

Equality of the probability distributions is equivalent to equality of the characteristic functions. Thus we demand for all $p, q \in \mathbb{R}$:

$$E\left[\exp\left\{i\begin{pmatrix}p\\ q\end{pmatrix}\left(\alpha\partial_x\begin{pmatrix}u\\ v\end{pmatrix} + \beta\partial_y\begin{pmatrix}u\\ v\end{pmatrix} + \gamma\begin{pmatrix}\Delta u\\ \Delta v\end{pmatrix}\right)\right\}\right] = E\left[\exp\left\{i\begin{pmatrix}p\\ q\end{pmatrix}\cdot\left(\alpha\begin{pmatrix}e\\ f\end{pmatrix} + \beta\begin{pmatrix}g\\ h\end{pmatrix}\right)\right\}\right].$$

Evaluating these expected values and separately collecting the coefficients of p^2, $2pq$, and q^2 we find that the equality of the characteristic functions is equivalent to the following:

$$e^2 + g^2 = A := (\partial_x u)^2 + (\partial_y u)^2 + (\Delta u)^2,$$

$$ef+gh = B := \partial_x u\,\partial_x v+\partial_y u\,\partial_y v+\Delta u\,\Delta v,$$
$$f^2+h^2 = C := (\partial_x v)^2+(\partial_y v)^2+(\Delta v)^2.$$

These are three equations for the four unknowns e, f, g, h. To compute a solution in a numerically stable way, we test whether $A > C$. If so, we set

$$e=\sqrt{A},\quad f=B/e,\quad g=0,\quad h=\sqrt{C-f^2}, \tag{7}$$

else

$$h=\sqrt{C},\quad g=B/h,\quad f=0,\quad e=\sqrt{A-g^2}. \tag{8}$$

5. Shader-based implementation

5.1. General approach

Implementation prototypes have been built in C# using Microsoft®'s Managed DirectX® programming interface; shaders were developed in HLSL and stored in an .fx file for Microsoft®'s Effect framework. A standard texture with no special preparation is used. To ensure a gamma-correct mapping of the averaged colors to the screen, we set the `SrgbWriteEnable` property of the DirectX® device to true.

The mesh data has to be changed to store additional static data to be computed upfront: Every vertex is equipped with the following data:

- its 3D position $\mathbf{p}$ in object space,
- a normal vector $\mathbf{n}$ in object space,
- its texture coordinates u and v,
- a matrix U according to Equation 5 as a pair of three-component vectors.

Inside the vertex shader, we transform the position $\mathbf{p}$ to camera space by forming $\mathbf{r} := M\mathbf{p}+\mathbf{v}$ using the 4×4 matrix `WorldView` offered by the Effect framework. Furthermore, the frame-to-frame difference of this matrix is formed to compute $\mathbf{s} := \Delta M\mathbf{p}+\Delta\mathbf{v}$. According to Equations 3, 4, and 5, we eventually have to determine

$$\begin{pmatrix}\Delta u\\ \Delta v\end{pmatrix} = U\Delta\mathbf{p} = -UM^{-1}\mathbf{s}+\frac{(M^{-1\mathrm{T}}\mathbf{n})\cdot\mathbf{s}}{(M^{-1\mathrm{T}}\mathbf{n})\cdot\mathbf{r}}UM^{-1}\mathbf{r}.$$

To allow the user to control the amount of motion blur applied, we multiply $\begin{pmatrix}\Delta u\\ \Delta v\end{pmatrix}$ by a parameter representing exposure time.

5.2. Pixel-based implementation

In the first implementation, we aim at a computation that remains precise for large triangles. The original 3D mesh is converted into a triangle list. For every triangle we store all three vertices, even though vertices with equal positions in 3D space may already have been stored for other triangles. This allows to store triangle normals (see Equation 1) instead of averaged normals. The matrix U is treated similarly.

In the vertex shader, we only compute terms that vary linearly, thus making efficient and precise use of the automatic bilinear interpolation taking place between vertex shader and pixel shader. These terms are: $-UM^{-1}\mathbf{s}$, $(M^{-1\mathrm{T}}\mathbf{n})\cdot\mathbf{s}$, $(M^{-1\mathrm{T}}\mathbf{n})\cdot\mathbf{r}$, and $UM^{-1}\mathbf{r}$. The pixel shader only needs to combine these quantities using one floating-point division and one multiply-and-add operation on vectors.

In the pixel shader, we apply the standard HLSL functions `ddx` and `ddy` to find the partial derivatives of u and v. The area over which to form the anisotropic average can now be found using Equations 7 and 8. e, f, g, h are directly inserted as parameters into the `tex2D` call of HLSL.

The vertex shader compiles to 29 assembly language instructions, the pixel shader to 30. The larger part of the pixel shader is devoted to the blending of spatial and temporal anisotropy. Temporal anisotropy alone could be dealt with in as few as five assembly language instructions.

5.3. Vertex-based implementation

The second implementation places as few as possible instructions in the pixel shader, relying on a fine tessellation. Here, we use the original structure of the mesh with vertices shared among adjacent triangles. The per-vertex normal $\mathbf{n}$ and the per-vertex matrix U are formed by averaging over their values for the adjacent triangles.

On first sight, it may appear natural to compute e, f, g, h in the vertex shader. However, there is one open degree of freedom in the computation of these quantities, see Section 4. We have to prescribe a rule that fixes this, such as requiring $g = 0$. It seems hard if not impossible to give a rule that leads to a meaningful linear interpolation applied to the vertices by the graphics chip. For instance, false near-zero results of the linear interpolation may result if the sign of a value changes one from vertex to a neighbor.

Therefore, we compute quantities in the vertex shader that are better suited for linear interpolation:

$$a := \sqrt{A},\quad c := \sqrt{C},\quad b := B/(ac),\quad d := \sqrt{1-b^2}.$$

From these, the pixel shader derives e, f, g, h. For numerical stability, we again distinguish two cases according. If $A > C$, we set $e = a$, $f = bc$, $g = 0$, $h = cd$, else $e = ad$, $f = 0$, $g = ab$, $h = c$.

Vertex shaders cannot use the HLSL functions `ddx` and `ddy`. To compute the screen-space partial derivatives of u and v we have to resort to matrix computations: A derivation similar to Equations 3, 4, and 5 leads to

$$\partial_x\begin{pmatrix}u\\ v\end{pmatrix} = UM^{-1}\left(r_z\mathbf{e}_x-\frac{(M^{-1\mathrm{T}}\mathbf{n})\cdot\mathbf{e}_x}{(M^{-1\mathrm{T}}\mathbf{n})\cdot\mathbf{r}}\mathbf{r}\right),$$

where r_z is the z component of $\mathbf{r}$, the vector $\mathbf{e}_x$ is defined by $(\frac{2}{fW},0,0)^{\mathrm{T}}$ with W being the screen width in pixels and f

denoting focal length, which can be read off from the `Projection` matrix. A similar formula applies to the partial derivative with respect to screen-space y.

The remaining computations are similar to the first implementation. In total, the vertex shader compiles to 53 assembly language instructions, the pixel shader to 7.

5.4. Reference implementation of multiple rendering

To have a standard for comparisons, we also implemented motion blur based on temporal supersampling: Images that correspond to N different instants of time are blended.

A way to re-render geometry without costly state changes is "instancing" as offered by Shader Model 3.0. The vertex buffer of the original 3D mesh forms the *instanced* stream. Here we employ the original, indexed mesh with shared vertices, not the augmented one constructed for the other implementations. The *instancing* stream contains one floating point value t for each of the N instances. It ranges from $-\frac{1}{2}$ to $\frac{1}{2}$ in uniform increments. The vertex shader computes $\mathbf{p} \mapsto M\mathbf{p} + \mathbf{v} + t\left(\Delta M\mathbf{p} + \Delta\mathbf{v}\right)$ to apply a gradually changing transformation to the different instances of the mesh.

To keep the reference implementation fast, we assume that the 3D model can be rendered without help of the depth buffer, for instance by using backface culling. Thus, we switch off writing to the depth buffer and rely fully on additive alpha blending. The rendering is done on a black background; colors are multiplied by $\frac{1}{N}$.

6. Results. Discussion

We used a PC equipped with an Intel® Pentium®-4 CPU running at 2.5 GHz and an NVIDIA® GeForce™ 6800 GT graphics card to evaluate speed and quality. In the graphics driver, we switched on the optimization for trilinear mapping, which led to a speedup by up to 20 % without objectionable losses in image quality. The additional switch for optimization of anisotropic filtering did not show any noticeable effect. For the multiple rendering method, we employed a maximum anisotropy degree of 4 throughout.

The artifacts generated by the novel method look different from those of multiple rendering: Where the latter starts to reveal multiple exposures, the former starts to blur excessively. As the streaks of motion blur get longer, the footprint to average over in the texture tends toward an increasingly long line in *uv* space. When this footprint can no longer be assembled from a reasonable number of MIP texture requests, the graphics chip resorts to coarser MIP levels. On top of that, the locally linear approximation of the motion can smear texels to screen pixels they never reach in reality.

Visually, the amount of artifacts of both implementations of the novel method with a maximum anisotropy degree of 8 corresponded to the artifacts produced by 16-fold multiple rendering, see Figure 6. This can be attributed to an intelligent assembling of the filter footprint by the graphics chip.

Figure 6 also shows that—as expected—the pixel-based implementation works well with large triangles, whereas the vertex-based implementation does not. On the other hand, the latter is better suited for fast motion of non-planar geometry due to the following: The pixel-based implementation treats each triangle on its own, assuming that the triangle is part of a plane with correspondingly continued *uv* parametrization. A fast motion can reveal that this assumption is not true: It leads to motion blur that extends far beyond the current triangle in *uv* space.

Speed benchmarks, see Figure 7, were done fullscreen on 1280×1024 pixels using no vertical synchronization. Naturally, the rendering is strongly fillrate-limited, as can be seen from the fact that the two models used differ by a factor of 40 in their triangle count, but only by a factor of less than 2 in rendering speed. The exposure time has a major impact on speed: Long exposure leads to a large degree of anisotropy. This increases the number of texture requests the graphics chip must issue to approximate the footprint. Thus, the `tex2D` call in the shader becomes the limiting step.

In total, the vertex-based and the multiple-rendering implementation yield similar speed and quality if the tessellation is fine enough for the linear interpolation used in the vertex-based method. Multiple rendering in the efficient form employed here reduces the number of colors, what shows as blotches. This is due to the division of the color by the number of passes. To overcome this, one would have to use an intermediate buffer of extended color range.

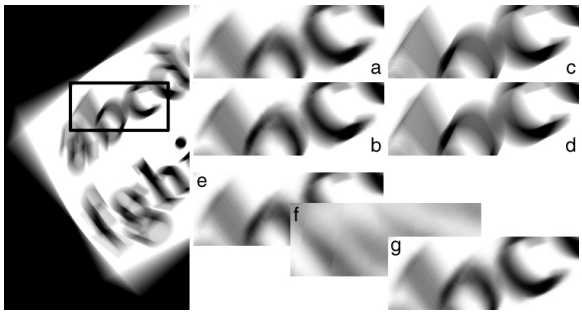

Figure 6: *A rotating billboard is used to illustrate the artifacts introduced by the different approaches. a, b: pixel-based method with a maximum anisotropy degree of 8 and 16; c, d: 16- and 32-fold multiple rendering. Whereas the vertex-based method works well with a tessellation of 800 triangles (e), it cannot handle a tessellation of only 2 triangles (f), in contrast to the pixel-based method (g).*

7. Summary. Outlook

We have presented a method to generate motion blur with help of filtered textures, leveraging the anisotropic filter-

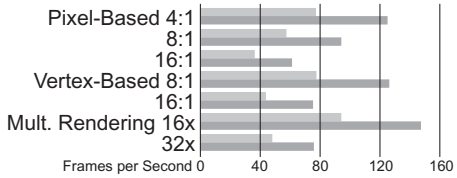

Figure 7: *To evaluate the speed, we used a flight over a landscape of 20,000 triangles (light gray) and a spinning sphere of 528 triangles (dark gray). The numbers refer to the maximum degree of anisotropy and to the count of multiple renderings. For details, see text.*

ing offered by current hardware. The method achieves results that can be compared in terms of quality and speed to a very efficient, non-general implementation of multiple rendering. Neither the texture-based motion blur introduced here nor the fast implementation of multiple rendering can handle occlusion in full generality. In that, they resemble most other methods for real-time motion blur such as [AW02] and [Gre03].

At the same time, the novel method avoids some issues of multiple rendering and other methods described in the literature. For instance, there is only one rendering pass. This avoids back-to-front sorting problems and drastically speeds up lighting and shading computations, which may be done in addition to motion blur. On top of that, there is no need and no time expense for an additional buffer with extended range or floating-point blending capabilities. The novel method will strongly profit from future progress in hardware support for adaptive anisotropic filtering.

The presented technique works well for a range of objects including billboards as well as terrains that do not contain strongly peaked mountains. In general, silhouettes will not be subjected to motion blur. However, objects that are cookie-cut from large polygons by alpha-blending show physically correct blurring of the silhouettes. Complex objects could be handled by combining a geometric approach with the texture-based motion blur, comparable to [Gre03]. On top of that, deforming meshes could be handled by integrating the intrinsic motion into Equation 2.

References

[ATI04] ATI: ATI Radeon X800: High definition gaming. White Paper, 2004. http://www.ati.com/products/radeonx800/HighDefinitionGamingWhitePaper.pdf.

[AW02] ARCE T., WLOKA M.: In-game special effects and lighting. GDC 2002 presentation, 2002. http://developer.nvidia.com/object/gdc_in_game_special_effects.html.

[CPC84] COOK R. L., PORTER T., CARPENTER L.: Distributed ray tracing. *Computer Graphics (Proc. SIGGRAPH '84) 18*, 3 (1984), 137–145.

[GH86] GREENE N., HECKBERT P.: Creating raster omnimax images from multiple perspective views using the elliptical weighted average filter. *IEEE Computer Graphics and Applications 6*, 6 (1986), 21–27.

[Gre03] GREEN S.: Stupid OpenGL shader tricks. GDC 2003 presentation, 2003. http://developer.nvidia.com/docs/IO/8230/GDC2003_OpenGLShaderTricks.pdf.

[HA90] HAEBERLI P., AKELEY K.: The accumulation buffer: Hardware support for high-quality rendering. *Computer Graphics (Proc. SIGGRAPH '90) 24*, 4 (1990), 309–318.

[Har04] HARGREAVES S.: Detail texture motion blur. In *ShaderX^3, Advanced Rendering with DirectX and OpenGL*, Engel W., (Ed.). Charles River Media, 2004, pp. 205–214.

[JK04] JONES N., KEYSER J.: Real-time geometric motion blur for a deforming polygonal mesh. In *Eurographics/ACM SIGGRAPH Symposium on Computer Animation; Posters and Demos* (2004).

[KB83] KOREIN J. D., BADLER N. I.: Temporal antialiasing in computer generated animation. *Computer Graphics (Proc. SIGGRAPH '83) 17*, 3 (1983), 377–388.

[MPFJ99] MCCORMACK J., PERRY R., FARKAS K. I., JOUPPI N. P.: Feline: Fast elliptical lines for anisotropic texture mapping. In *Proc. SIGGRAPH '99* (1999), pp. 243–250.

[Nvi03] NVIDIA: NVIDIA GeForce FX GPUs: Intellisample technology. Technical Brief, 2003. http://www.nvidia.com/object/intellisample_tb.html.

[Nvi04] NVIDIA: Spinfx.fxproj. FX Composer Example, 2004. http://developer.nvidia.com/developer/SDK/Individual_Samples/effects.html.

[PC83] POTMESIL M., CHAKRAVARTY I.: Modeling motion blur in computer-generated images. *Computer Graphics (Proc. SIGGRAPH '83) 17*, 3 (1983), 389–399.

[SLK01] SHIN H.-C., LEE J.-A., KIM L.-S.: SPAF: Subtexel precision anisotropic filtering. In *Proc. HWWS '01* (2001), pp. 99–108.

[SPW02] SUNG K., PEARCE A., WANG C.: Spatial-temporal antialiasing. *IEEE Image Processing and Applications 8*, 2 (2002), 144–153.

[TBI03] TATARCHUK N., BRENNAN C., ISIDORO J.: Motion blur using geometry and shading distortion. In *ShaderX^2: Shader Programming Tips and Tricks with DirectX 9*, Engel W., (Ed.). Wordware, 2003, pp. 299–308.

[WZ96] WLOKA M. M., ZELEZNIK R. C.: Interactive real-time motion blur. *The Visual Computer 12*, 6 (Sept. 1996), 283–295.