Seventh Hungarian Conference on Computer Graphics and Geometry, Budapest, 2014

Realtime, coherent screen space hatching

Zoltán Lengyel,[1] and Tamás Umenhoffer[1] and László Szécsi[1]

[1] Budapest University of Technology and Economics, Department of Control Engineering and Information Technology

Abstract

In this paper we present a screen space hatching algorithm which provides time coherent placing of hatching lines relative to object surfaces. While with screen space techniques we can easily achieve consistent image space hatching density, it is hard to make hatching lines express surface features, and to make them follow the underlying geometry. Drawing individual textured lines can provide high quality results, but their direction and amount of bending should be calculated according to the 3d geometry. We propose a method that combines illumination gradient, and curvature based line direction calculation to support a wide variety of objects. To achieve surface position coherency during animation we use image space velocity maps to move the individual hatch lines, and use rejection sampling and low discrepancy sequences to filter out high density areas where the flow accumulates lines, and fill in the vacant areas. The multi-pass algorithm is implemented entirely on the GPU using geometry shaders and vertex transform feedback.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

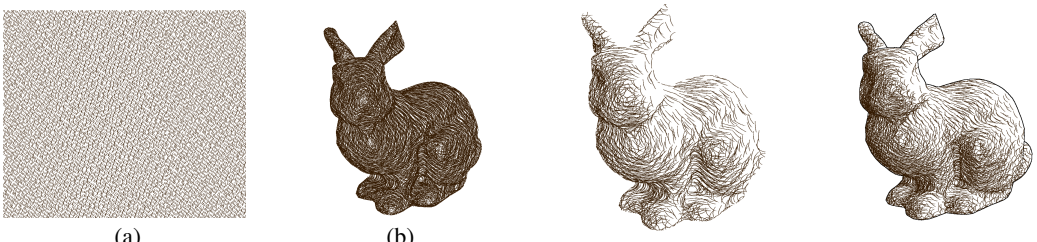
1. Introduction

A wide range of techniques in computer graphics target non-photorealistic (NPR) or illustrative image production, among them the synthesis of hand-drawn art. Like photorealistic techniques, NPR techniques are also based on three dimensional objects, but try to mimic the look of traditional media. The 3D modelling and rendering pipeline has great efficiency benefit compared to handmade image generation, but it is very hard to reproduce the visual look of classic techniques. 3D rendered images have a restricted freedom regarding geometry borders, smooth lighting transitions and surface materials. Though several techniques exists, that can well mimic special traditional techniques, their applicabilities are limited.

Animation movies focus less on rendering speed but on rendering quality, but this does not mean that render time is not important. Increased rendering time can have serious cost influences, thus NPR techniques made for offline rendering should also keep processing time low. An other important aspect is that preprocessing requirements should also be treated carefully, as in a production environment the modelling, the rendering, and the final compositing is done in separate packages, which have their own specialities, thus sharing data between them can be very hard. Huge preprocessing or per-frame geometry processing is also not feasible for realtime applications.

The biggest limitation of most NPR techniques both for realtime and for production purposes is time coherency. It is hard to ensure that stylized lines, illustrative surface details or lighting features follow the geometry during animation. If these features are fixed in image space it produces an annoying artifact called *shower door* effect, as it seems like the NPR effect is caused by a distorting glass door we are looking through. These artistic features should move with the geometry without any sudden change or flickering.

In this paper we focus on pen and ink illustration and pencil drawings, where the lighting and shadows and the shape of the objects are represented with the density and orientation of thin hatch lines. To mimic the traditional 2D technique we should ensure that hatching line density, line length and width is consistent in screen space. Hatch lines should be placed more densely in shadowed areas, and they should be completely missing in lit areas. Also their orientation and bending should well describe the underlying geometry.



(a) (b) (c) (d)

Figure 1: *Basic elements of the hatching rendering algorithm from left to right: hatch line positions in screen space (a), rotated and bent lines (b), rejected lines according to lighting (c), adding edge detected contours (d).*

2. Previous work

The most obvious way to add artistic look to our objects is to paint an artistic texture for them. This also works for hatching: we can use an image of hatch lines and map this as a repeated texture to our objects. Choosing the detail of this image is not as simple as in case of usual textures. As we move closer to the surface more lines should appear. Consistent image space line density can be achieved with mipmapping, multiple texturing and texture blending [6 4 5 9 8 12]. One drawback is that the maximum detail is limited as the number of mipmap levels and texture resolution is also limited. The other problem is that a proper parametrization is needed, and not only to avoid texture seams, but to make the lines follow geometry features. Manual UV layout creation is rather time consuming, and seams can not be eliminated completely. Automatic methods need to calculate principal curvature directions from the geometry and orient smaller hatching patches usually placed in screen space.

The another main group of techniques generates new geometry for hatch lines. Here we distinguish between object space and image space methods. Object space methods place hatching lines directly onto the rendered surface in 3D space [11 1 3]. The lines are rendered as polygon strips. Principal curvature directions should also be calculated to make the lines follow the surface. The strength of these methods is that lines are tied onto the surfaces providing straightforward temporal coherence. On the other hand visibility calculation of the lines can cause biasing problems, and it is also hard to ensure uniform distribution of hatching lines in image space. With these techniques lines go through the same transform pipeline as all other rendered polygons.

Hatching lines can also be drawn in image space [2 10]. These techniques work with uniformly placed hatch lines in screen space. Determining the direction of the lines needs special considerations. The lines should illustrate the underlying surface, thus a proper image space directional field should be created. Different approaches use different quantities to calculate this vector field. They might use the tone gradient of an input image, or use screen space principal curvature direction data. The later can be calculated in screen space if some necessary information like camera space depth or normal vectors can also be rendered. Principal curvature directions could also be computed from the processed geometry and projected onto the image plane, but this requires geometry processing. Image space methods often suffer from temporal incoherence.

3. Motivation

Our goal is to develop a hatching rendering technique that defines individual lines in screen space, but keeps time coherency so no flickering or sudden change appears. Lines should have an even distribution in screen space, moving closer to objects should make more hatching lines appear. Line direction and curving should well express surface features, they should accurately fit onto the 3D surface. Animation should be supported with moving the lines with the objects in screen space but still maintaining even screen space density.

We should avoid complex geometry processing, moreover we should make our algorithm independent of the geometry, and rely only on standard outputs of common renderers like image buffers. This enables us to implement the algorithm in an interactive post processing framework. As the main goal is to use the technique interactively all calculations should be kept on the GPU.

4. Proposed algorithm

Our technique is based on [7]. Hatching lines are rendered as individual triangle strips. Even density of hatching lines can be easily achieved with placing the lines randomly on screen using a uniform distribution. Illumination can be depicted with filtering out particles at highlight areas. Figure 1 shows the main components of screen space hatching generation.

Our main contribution to the base work is to support time coherent line animation. Lines are moved according to a velocity map, which results an uneven distribution. To fix the uneven particle density we filter out dense regions and fill in sparse regions. The final steps are to calculate hatching direction and blending for the particles and render them as

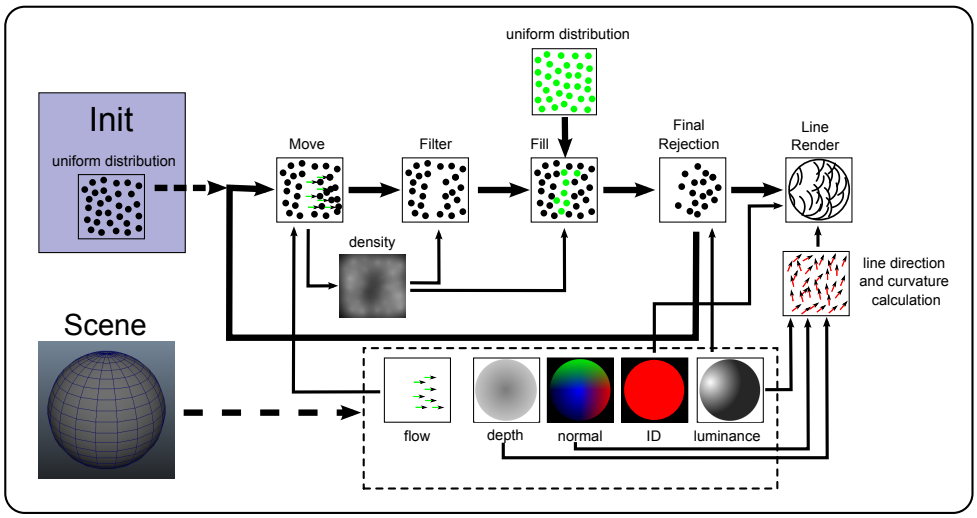


Figure 2: *Basic elements of the proposed algorithm. In initialization phase uniform screen space particle density is generated. In each frame buffers containing normals, depth, illumination, object ID, and screen space velocity are rendered. Particles are advected according to the velocity. The resulting density is not uniform which will be equalized with a filtering process followed by inserting in new random particles. Final screen space density is usually defined by a luminance map, so further particles are rejected. As a last step lines are drawn as textured line primitives.*

textured line srtips. The final particles after the fill and filter process will be the input of the next rendered frame. The necessary information for the algorithm is several buffers rendered in each frames. These buffers store such data that is common both in realtime and in production environments: depth map, normal map, illumination, id map and velocity map. Figure 2 shows the main components of the proposed algorithm.

Our second contribution new to [7] is that we propose an automatic hatching direction calculation that combines several surface features. The following subsections cover the main components of the algorithm in details.

4.1. Hatching particle generation

The first step is to define a set of particle positions (seed points) in screen space, at which final lines will be rendered. These particles should have a uniform distribution to evenly cover the image plane, and they should show no recognizable pattern to mimic the random behaviour of hand drawn hatch lines. This can be easily achieved with placing the lines randomly on screen using a uniform distribution. Artist-defined global density can be implemented with fewer or more random samples on screen.

Random particle generation is the initialization phase of our algorithm, where we create a huge buffer of random positions using Halton sequences. The size of this buffer is much bigger than the desired line number count as later, during animation more and more random positions will be requested. If we ran out of the buffer, we start from the beginning. For each particle we also store an additional random number from the unit interval which we call priority. Priorities should also have a uniform distribution in image space. This can be easily achieved using low-discrepancy series like Halton sequences with using the normalized sequence number as priority, but as we use only a part of the buffer and need priorities from the whole unit interval, it was easier to define an additional random sequence for priorities too. For smoother animation we can also store additional line data for each particle like line length, width, direction or curvature.

4.2. Particle movement

As the camera or any objects in the scene move, we should ensure that the seed positions will move with the surface. This is crucial to avoid the unwanted shower door effect. As we are working in image space, the most obvious solution is to use a screen space velocity map, or in other worlds: an optical flow. As our algorithm is based on 3D scenes we can produce this map quite easily and accurately. We can use the camera matrices and the world matrices for each object from the previous frame, transform the vertex positions with both the new and the previous transformations from which movement can be calculated with a simple vector difference.

Using this flow map particles are advected. After moving particles, some will remain in its original position, some will move to an other image position and some will fall outside the screen. For most of the movements this results an uneven particle distribution.

4.3. Filtering dense areas

After particle movement we should restore the even artist-defined particle density. Our first step is to filter out too dense areas. This filtering means that we should keep a particle only if drawing that particle will not make its local neighbourhood too dense. Refreshing density during each particle draw is not an option in a real-time environment as dependency between the rendering of each line can make parallel hardware implementation impossible. To overcome this we should make the rejection of a single line dependent only on a local desirable density but not on the influence of previously rendered lines. To achieve this we use the theory of rejection sampling.

The classic rejection sampling problem is when we would like to achieve a desired distribution, but we can not use the inversion method. With rejection sampling we choose a well known distribution which is easy to create and upper bounds the desired distribution. The simplest distribution to use is the uniform distribution with an appropriate scale. If we assign a random priority for each sample of the uniform distribution and reject the sample if its priority is above the desired distribution, the remaining samples will have the desired distribution.

On our specific case the desired distribution is a uniform distribution and our particles have an uneven distribution which locally exceeds the desired uniform distribution. To filter out the particles we should know the density at the neighbourhood of each particle. To do this we render a small disk shaped snippet at each particle with a linear falloff and blend them together with additive blending. The result is a bit noisy density function. The radius of the snippets and their power defines the locality and smoothness of the density map, these parameters are set empirically. Note that these parameters should depend on the resolution and the number of desired hatching lines. After parameter tuning we used the following expressions which worked well for different resolutions and line numbers:

$$snippet\_size = \sqrt{\frac{number\_of\_pixels}{desired\_line\_number}} * \frac{6.5}{window\_resolution}$$
$$snippet\_power = 0.04 * linearFalloff()$$

If the density map is given we examine each particle and reject them if their priority scaled with the underlying density is above the desired uniform density. As priorities are uniformly distributed, lines will be rejected uniformly thus the resulting density in high density areas will also match the original uniform distribution. After this filtering step particle priorities should also be mapped back to the unit interval, as high priority particles were filtered out. This will ensure the uniform priority distribution again.

4.4. Refill sparse areas

After the filtering process we still need to fill in vacant areas. To do this we take an other set of random samples and try to fill the image with them. Here we also use the former density map and keep particles only if their priority is above the underlying density value. Thus at coarser areas we keep more particles, and where we already reached the desired density we reject all new particles. Again, particle priorities should be mapped back to the unit interval. The new particles are placed at the end of the former filtered particle buffer. At this stage we again have a uniform particle position and priority distribution in image space, but particles are moving with the surfaces wherever it is possible, which makes the illusion of being defined in object space.

4.5. Illumination

Hatching density also depicts current lighting conditions, thus illuminated image regions should have coarser hatching density than areas in shadows. Here again we face the classic problem of rejection sampling. Our desired distribution is the illumination value, and we have a uniform distribution to reject samples from. Each particle, whose priority is below the inverted luminance value will be rejected. At this step particles are not removed only their rendering is passed, thus they will stay in the particle buffer which will be the input of the next frame.

We can make animation more smooth if we don't let sudden disappear and appear of lines due to luminance change or flow filtering. We can store the line length for each particle, decrease this length if the particle is marked for removal and remove it only if its length reached zero. The same can be used for new lines: they start with a small line length and in each frame their length will be increased. For smooth luminance change we don't even need to store previous length values they can be calculated on the fly by applying a smoothstep function centred at the rejection priority cutoff value. Note that original solution is equivalent with a step function at the inverted luminance value.

4.6. Line direction and bending calculation

Before final rendering the lines, we need to define their direction and amount of bending. These features should be chosen in a way that they describe the underlying geometry well, and mimic the way an artist would orient them. Due to our experiments presented in [7], we can say that no single feature exists that can be used for all types of geometry (see figure 3). On the other hand principal curvature directions are commonly used in NPR techniques to orient lines, and artist also find it as a natural orientation.

Principal curvature can be calculated from the curvature tensor which is the Hessian of the depth field. As surface normals describe the depth gradient the Hessian matrix can be defined in terms of the directional derivatives of the surface normal:

$$H = \begin{pmatrix} \frac{\partial \vec{n}}{\partial \vec{u}} \cdot \vec{u} & \frac{\partial \vec{n}}{\partial \vec{v}} \cdot \vec{u} \\ \frac{\partial \vec{n}}{\partial \vec{u}} \cdot \vec{v} & \frac{\partial \vec{n}}{\partial \vec{v}} \cdot \vec{v} \end{pmatrix}$$

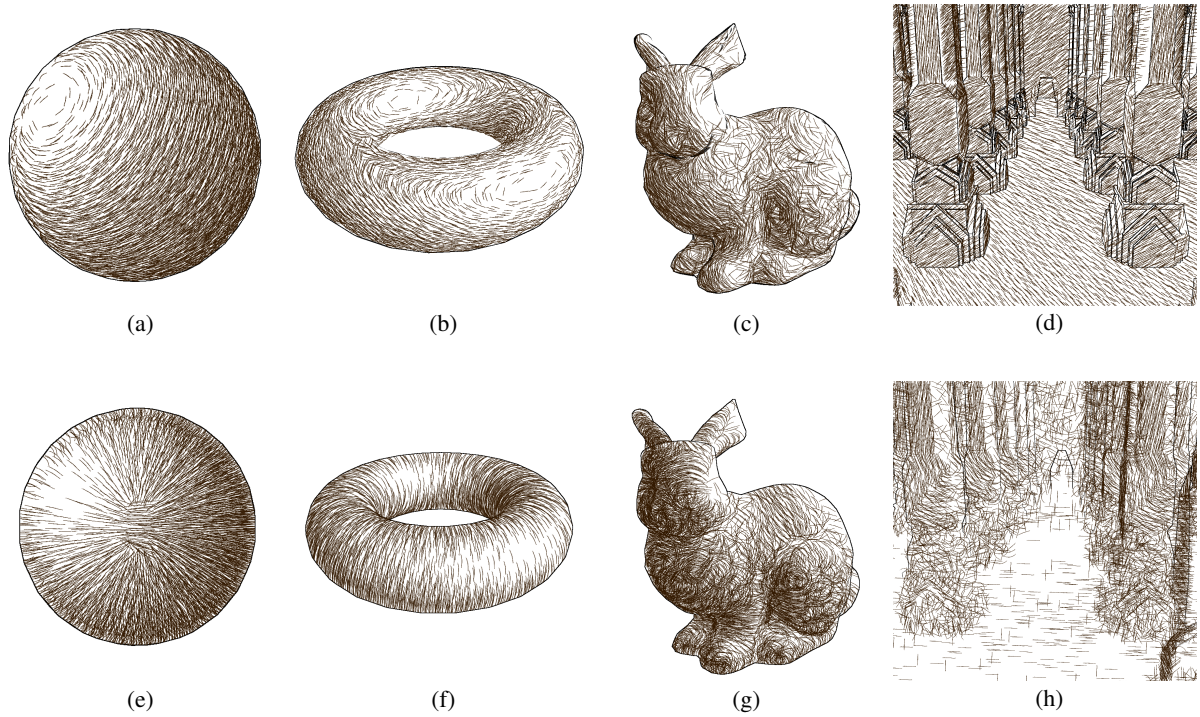


Figure 3: *Hatching using luminance (top) and principal curvature features (bottom). Different geometry types need different approaches to represent main surface features.*

where $\vec{u}$ and $\vec{v}$ are orthonormal tangent vectors on the surface. In our case, they are the projections of the screen space unit $\vec{x}$ and $\vec{y}$ vectors to the surface. In other words, the Hessian is the screen space directional derivatives of the screen space normal vectors. The gradient is computed with a Sobel filter and its value is compensated with the projected length of the surface normal. The eigenvalues and eigenvectors of this matrix define the maximum and minimum normal curvature values and their corresponding directions, thus the principal curvature and principal curvature direction.

We should note that using the normal vectors as first order depth gradients will lead to clearly visible tessellation in the second order features, as normal vectors are only linear approximations of a smooth surface. In some cases this is does not produce obvious artifacts as lines are not placed too dense to make these sudden changes annoying. On the other hand when moving closer to the surface the relative density gets higher, thus these triangle borders will be visible. We also found that during the animation of the particles, some of them can randomly move across of one of these borders, which makes flickering orientation changes. To overcome these problems we used an edge preserve smoothing filter on the normal vectors before derivation.

In our tests we found that luminance gradient is also a good feature that can describe the geometry well, and can handle some special cases that principal curvature directions not. These cases include surfaces with no curvature like flat geometry, or with no principal curvature like a sphere. On the other hand some geometry like a torus can be better described with principal curvature directions (see figure 3).

To combine the advantage of both methods we calculate both features and use the luminance gradient based line direction where principal curvatures are uncertain. These are the cases when the maximal and the minimal curvature equals, and where no principal curvature can be calculated. Figure 4 shows our automatic feature selection technique on different type of geometries.

4.7. Final render

After line direction and amount of bending is calculated lines are drawn as curved and textured triangle strips with hardware blending enabled. The lines can be drawn directly to screen with orthographic projection, as image space samples do not need any 3D transformations, nor visibility testing. Additionally we can clip lines on a per pixel basis using an ID map to prevent them from crossing object borders. This behaviour is useful in case of longer lines, but shorter lines can even pass these borders without visible artifacts, which makes the final rendering even more natural and hand drawn like.

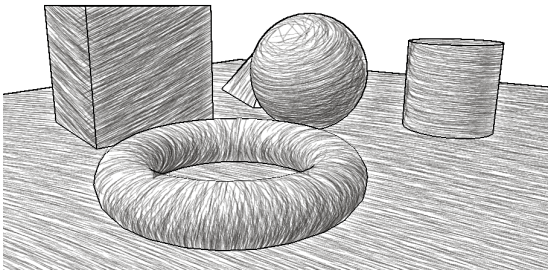


Figure 4: *Line direction and curving calculated on a per line decision between luminance and principal curvature features.*

5. Implementation

We implemented the hatching synthesis algorithm in a standalone application using OpenGL and GLSL shaders. Geometry buffers like depth and normal maps are rendered in each frame, and the proposed algorithm uses them as input. Note that this implementation can also be extended to support input from an image stream rendered with an offline renderer. Image processing of the buffer inputs like gradient calculation, principal curvature calculation or blurring was implemented with full screen quad fragment shaders and render to texture.

The main work flow of the algorithm is to process a dynamic array of particle data. This data processing can be implemented with geometry shaders. The input of the shaders are point primitives storing the particle data. A geometry shader instance processes one particle and alters this data if necessary, or it can completely reject it. The output of the geometry shader is also a point primitive, which can be sent to the rasterizer, or in our specific case it can be fed back to an other buffer on the GPU without actual drawing. Output feedback can be directed to a specific position in the GPU buffer, so merging of two buffers is also possible (this is needed in the particle refill phase). The number of lines that was written to the buffers can be queried with the OpenGL API, so we can always know the actual valid line count in our buffers.

To get the actual density map, a geometry shader extrudes small quads from each particle, and the pixel intensities are calculated in the fragment shader. During final line rendering a geometry shader creates bended line strips from the particle points and sends them to the rasterizer. Our implementation keeps all calculation and necessary data on the GPU from input data processing to final display.

6. Results

As our algorithm works completely in screen space the performance does not depend on the actual geometry only on screen resolution. We should note that creating the input buffers need additional scene rendering passes thus the performance is afterall influenced by geometry complexity. This can be eased with the use of multiple render targets to output all necessary information in a single render pass. The main limitation factors are line count and screen resolution. We found that creating the density map is a critical part of the algorithm, as because of the blending a serious amount of pixel overdraw is present. This can be eased with choosing a smaller snippet radius, which makes the density estimation more local and bit more noisy. On the other hand using more lines does not increase the computational cost of this density rendering phase as more lines results in smaller snippet size which reduces the number of fragments processed (see the second column in table 1).

Choosing the snippet size rightly also influences line flickering as we can not create a completely uniform density map with drawing snippets, thus even without particle movement some areas will be filtered while others will be filled with extra particles. This results in a continuous disappear and reborn of particles even within a static environment which is an unwanted effect. To handle this we introduced a threshold range around the desired density. Within this range we treat the area as having the desired density. This threshold value together with density snippet size and intensity should be fine tuned to get good results.

The other aspect that greatly decreased the performance is the edge preserve blurring we applied to the buffers before taking their gradient (see the third column in table 1). The cost of this this step depends on the blur kernel and the screen resolution. If little flickering is not a problem this step can be skipped.

Number of lines	FPS without filtering	FPS with filtering
10000	195	80
20000	190	80
40000	180	75
100000	145	60
200000	100	50

Table 1: *Performance tests on a Geforce GTX 480 with 1280x720 screen resolution.*

7. Conclusion

We presented a real-time hatching rendering algorithm that randomly places textured hatch lines on the image plane. Lines are advected according to screen space velocity of the

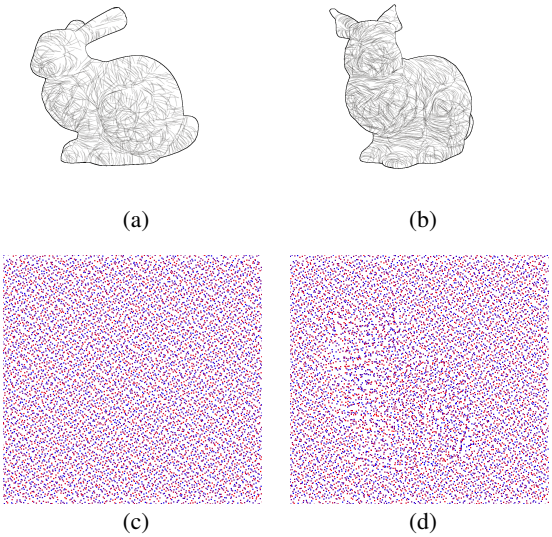


Figure 5: *Frames from an animated sequence. On the left the initial frame and its corresponding particle distribution is shown. On the right a later frame and its distribution is shown after rotating the camera. Uniform distribution is well preserved during animation. Some particle accumulation still occurs at object borders, but this is not visible on the final image.*

surfaces. Uneven image space particle density caused by particle movement is equalized using rejection sampling methods. Lines are rendered as textured line strips to the screen. The resulting algorithm eliminates the time coherency problem of screen space methods (see figure 5) but can keep their advantages like no visibility test is needed, uniform screen space particle density is maintained and the performance is independent of geometry complexity. Hatching lines can have a wide variety of styles by adjusting line density, width, length, maximal bending and applying artistic textures. Our GPU implementation provides real-time performance in high resolution environments.

Acknowledgements

This work has been supported by OTKA ????.

References

1. Gershon Elber. Interactive line art rendering of freeform surfaces. *Comput. Graph. Forum*, 18(3):1–12, 1999.
2. Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. In *PROCEEDINGS OF SIGGRAPH 2000*, pages 517–526, 2000.
3. Matthew Kaplan, Bruce Gooch, and Elaine Cohen. Interactive artistic rendering. In *Non-Photorealistic Animation and Rendering 2000 (NPAR '00)*, Annecy, France, June 5-7,2000.
4. Yongjin Kim, Jingyi Yu, Xuan Yu, and Seungyong Lee. Line-art illustration of dynamic and specular surfaces. *ACM Transactions on Graphics (SIGGRAPH ASIA 2008)*, 27(5), December 2008.
5. Adam Lake, Carl Marshall, Mark Harris, and Marc Blackstein. Stylized rendering techniques for scalable real-time 3d animation. In *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, NPAR '00, pages 13–20, New York, NY, USA, 2000. ACM.
6. Hyunjun Lee, Sungtae Kwon, and Seungyong Lee. Real-time pencil rendering. In Douglas DeCarlo and Lee Markosian, editors, *International Symposium on Non-Photorealistic Animation and Rendering (NPAR)*, pages 37–45. ACM, 2006.
7. Zoltán Lengyel, Tamás Umenhoffer, and László Szécsi. Screen space features for real-time hatching synthesis. In *Proceedings of the 9th conference of the Hungarian Association for Image Processing and Pattern Recognition*, KEPAF '13, pages 82–94, 2013.
8. Afonso Paiva, Emilio Vital Brazil, Fabiano Petronetto, and Mario Costa Sousa. Fluid-based hatching for tone mapping in line illustrations. *Vis. Comput.*, 25(5-7):519–527, April 2009.
9. Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-time hatching. In *In Proceedings of SIGGRAPH 2001*, pages 579–584. ACM Press, 2001.
10. Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. Interactive penâĂŞandâĂŞink illustration. In *In Proceedings of SIGGRAPH âĂŹ94*, pages 101–108, 1994.
11. Tamás Umenhoffer, László Szécsi, and László Szirmay-Kalos. Hatching for motion picture production. *Comput. Graph. Forum*, 30(2):533–542, 2011.
12. Johannes Zander, Tobias Isenberg, Stefan Schlechtweg, and Thomas Strothotte. High quality hatching. *Comput. Graph. Forum*, 23(3):421–430, 2004.